# CONTENTS

1 Organization 3

2 Animation 5
  2.1 AnimationInterface 5
  2.2 NetAnim 5

3 Ad Hoc On-Demand Distance Vector (AODV) 9
  3.1 Model Description 9
  3.2 Usage 11
  3.3 Validation 11

4 Applications 13

5 Bridge NetDevice 15

6 Click Modular Router Integration 17
  6.1 Model Description 17
  6.2 Usage 18
  6.3 Validation 20

7 CSMA NetDevice 21
  7.1 Overview of the CSMA model 21
  7.2 CSMA Channel Model 22
  7.3 CSMA Net Device Model 23
  7.4 Using the CsmaNetDevice 24
  7.5 CSMA Tracing 25
  7.6 Summary 26

8 DSDV Routing 27
  8.1 DSDV Routing Overview 27
  8.2 References 28

9 Emulation Overview 29
  9.1 Emu NetDevice 30
  9.2 Tap NetDevice 35

10 Energy Framework 41
  10.1 Model Description 41
  10.2 Usage 42
11 Flow Monitor

12 Internet Models
   12.1 Internet Stack ................................................. 47
   12.2 IPv4 ............................................................ 50
   12.3 IPv6 ............................................................ 51
   12.4 Routing overview ............................................... 51
   12.5 TCP models in ns-3 ........................................... 57

13 LTE Module .......................................................... 63
   13.1 Model Description ............................................. 63
   13.2 Usage ........................................................... 67
   13.3 Validation ....................................................... 68

14 Mesh NetDevice ...................................................... 69

15 MPI for Distributed Simulation .................................. 71
   15.1 Current Implementation Details ............................... 71
   15.2 Running Distributed Simulations .............................. 72
   15.3 Tracing During Distributed Simulations ..................... 74

16 Network Module ..................................................... 75
   16.1 Packets .......................................................... 75
   16.2 Node and NetDevices Overview ............................... 86
   16.3 Sockets APIs ................................................... 87
   16.4 Simple NetDevice .............................................. 90

17 Optimized Link State Routing (OLSR) ......................... 91
   17.1 Model Description ............................................. 91
   17.2 Usage ........................................................... 92
   17.3 Validation ....................................................... 92

18 OpenFlow switch support ....................................... 93
   18.1 Model Description ............................................. 93
   18.2 Usage ........................................................... 94
   18.3 Validation ....................................................... 96

19 PointToPoint NetDevice ........................................ 97
   19.1 Overview of the PointToPoint model ......................... 97
   19.2 Point-to-Point Channel Model ............................... 98
   19.3 Using the PointToPointNetDevice ......................... 98
   19.4 PointToPoint Tracing ....................................... 98

20 Propagation ........................................................ 101

21 Simple NetDevice ................................................ 103

22 Statistical Framework ........................................... 105
   22.1 Goals .......................................................... 105
   22.2 Overview ....................................................... 105
   22.3 To-Do .......................................................... 106
   22.4 Approach ........................................................ 106
   22.5 Example ........................................................ 107

23 UAN Framework .................................................... 115
   23.1 Model Description ............................................. 115
This is the ns-3 Model Library documentation. Primary documentation for the ns-3 project is available in five forms:

- ns-3 Doxygen: Documentation of the public APIs of the simulator
- Tutorial, Manual, and Model Library (this document) for the latest release and development tree
- ns-3 wiki

This document is written in reStructuredText for Sphinx and is maintained in the doc/models directory of ns-3’s source code.
This manual compiles documentation for ns-3 models and supporting software that enable users to construct network simulations. It is important to distinguish between modules and models:

- **ns-3** software is organized into separate modules that are each built as a separate software library. Individual ns-3 programs can link the modules (libraries) they need to conduct their simulation.
- **ns-3 models** are abstract representations of real-world objects, protocols, devices, etc.

An ns-3 module may consist of more than one model (for instance, the internet module contains models for both TCP and UDP). In general, ns-3 models do not span multiple software modules, however.

This manual provides documentation about the models of ns-3. It complements two other sources of documentation concerning models:

- the model APIs are documented, from a programming perspective, using Doxygen. Doxygen for ns-3 models is available on the project web server.
- the ns-3 core is documented in the developer’s manual. ns-3 models make use of the facilities of the core, such as attributes, default values, random numbers, test frameworks, etc. Consult the main web site to find copies of the manual.

Finally, additional documentation about various aspects of ns-3 may exist on the project wiki.

A sample outline of how to write model library documentation can be found in src/template/doc.

The remainder of this document is organized alphabetically by module name.

If you are new to ns-3, you might first want to read below about the network module, which contains some fundamental models for the simulator. The packet model, models for different address formats, and abstract base classes for objects such as nodes, net devices, channels, sockets, and applications are discussed there.
ANIMATION

Animation is an important tool for network simulation. While ns-3 does not contain a default graphical animation tool, we currently have two ways to provide animation, namely using the PyViz method or the NetAnim method. The PyViz method is described in http://www.nsnam.org/wiki/index.php/PyViz. The NetAnim method is described in detail at http://www.nsnam.org/wiki/index.php/NetAnim. We will describe the NetAnim method briefly here.

2.1 AnimationInterface

The class “AnimationInterface” under “src/netanim” uses underlying ns-3 trace sources to construct a timestamped ASCII file in XML format that can be read by a standalone animator named “NetAnim”.

2.1.1 Generating XML trace files for use in NetAnim

Apply the following statements before the “Simulator::Run ()” statement::

AnimationInterface anim (“animation.xml”)

where “animation.xml” is any arbitrary file name. It is important to ensure that your wscript includes the “netanim” module. Example as in: src/netanim/examples/wscript. Also include the header [#include “ns3/netanim-module.h”] in your test program.

The examples under “src/netanim/examples” illustrate this. The sample wscript is at “src/netanim/examples/wscript”. Lets take an example: “src/netanim/examples/star-animation.cc”. To run the example::

./waf --run "star-animation"

This will generate an xml file “star-animation.xml” in the same directory. This XML file contains the information required by the standalone animator “NetAnim” to produce the required animation.

2.1.2 Parts of the XML

This is described in detail at http://www.nsnam.org/wiki/index.php/NetAnim#Parts_of_the_XML

2.2 NetAnim

NetAnim is a stand-alone program which uses the custom trace files generated by the animation interface to graphically display the simulation. NetAnim is based on the multi-platform Qt4 GUI toolkit. A screenshot of the NetAnim GUI is shown below.
Figure 2.1: NetAnim GUI with dumbbell animation.
The NetAnim GUI provides play, pause, and record buttons. Play and pause start and stop the simulation. The record button starts a series of screenshots of the animator, which are written to the directory in which the trace file was run. Two slider bars also exist. The top slider provides a “seek” functionality, which allows a user to skip to any moment in the simulation. The bottom slider changes the granularity of the time step for the animation. Finally, there is a quit button to stop the simulation and quit the animator.

For detailed instructions on installing “NetAnim” and loading the XML trace file (mentioned earlier) using NetAnim please refer: http://www.nsnam.org/wiki/index.php/NetAnim
AD HOC ON-DEMAND DISTANCE VECTOR (AODV)

This model implements the base specification of the Ad Hoc On-Demand Distance Vector (AODV) protocol. The implementation is based on [rfc3561].

The model was written by Elena Buchatskaia and Pavel Boyko of ITTP RAS, and is based on the ns-2 AODV model developed by the CMU/MONARCH group and optimized and tuned by Samir Das and Mahesh Marina, University of Cincinnati, and also on the AODV-UU implementation by Erik Nordström of Uppsala University.

3.1 Model Description

The source code for the AODV model lives in the directory src/aodv.

3.1.1 Design

Class ns3::aodv::RoutingProtocol implements all functionality of service packet exchange and inherits from ns3::Ipv4RoutingProtocol. The base class defines two virtual functions for packet routing and forwarding. The first one, ns3::aodv::RouteOutput, is used for locally originated packets, and the second one, ns3::aodv::RouteInput, is used for forwarding and/or delivering received packets.

Protocol operation depends on many adjustable parameters. Parameters for this functionality are attributes of ns3::aodv::RoutingProtocol. Parameter default values are drawn from the RFC and allow the enabling/disabling protocol features, such as broadcasting HELLO messages, broadcasting data packets and so on.

AODV discovers routes on demand. Therefore, the AODV model buffers all packets while a route request packet (RREQ) is disseminated. A packet queue is implemented in aodv-rqueue.cc. A smart pointer to the packet, ns3::Ipv4RoutingProtocol::ErrorCallback, ns3::Ipv4RoutingProtocol::UnicastForwardCallback, and the IP header are stored in this queue. The packet queue implements garbage collection of old packets and a queue size limit.

The routing table implementation supports garbage collection of old entries and state machine, defined in the standard. It is implemented as a STL map container. The key is a destination IP address.

Some elements of protocol operation aren’t described in the RFC. These elements generally concern cooperation of different OSI model layers. The model uses the following heuristics:

- This AODV implementation can detect the presence of unidirectional links and avoid them if necessary. If the node the model receives an RREQ for is a neighbor, the cause may be a unidirectional link. This heuristic is taken from AODV-UU implementation and can be disabled.
• Protocol operation strongly depends on broken link detection mechanism. The model implements two such heuristics. First, this implementation support HELLO messages. However HELLO messages are not a good way to perform neighbor sensing in a wireless environment (at least not over 802.11). Therefore, one may experience bad performance when running over wireless. There are several reasons for this: 1) HELLO messages are broadcasted. In 802.11, broadcasting is often done at a lower bit rate than unicasting, thus HELLO messages can travel further than unicast data. 2) HELLO messages are small, thus less prone to bit errors than data transmissions, and 3) Broadcast transmissions are not guaranteed to be bidirectional, unlike unicast transmissions. Second, we use layer 2 feedback when possible. Link are considered to be broken if frame transmission results in a transmission failure for all retries. This mechanism is meant for active links and works faster than the first method.

The layer 2 feedback implementation relies on the `TxErrHeader` trace source, currently supported in AdhocWifiMac only.

### 3.1.2 Scope and Limitations

The model is for IPv4 only. The following optional protocol optimizations are not implemented:

1. Expanding ring search.
2. Local link repair.
3. RREP, RREQ and HELLO message extensions.

These techniques require direct access to IP header, which contradicts the assertion from the AODV RFC that AODV works over UDP. This model uses UDP for simplicity, hindering the ability to implement certain protocol optimizations. The model doesn’t use low layer raw sockets because they are not portable.

### 3.1.3 Future Work

No announced plans.
3.1.4 References

3.2 Usage

3.2.1 Examples

3.2.2 Helpers

3.2.3 Attributes

3.2.4 Tracing

3.2.5 Logging

3.2.6 Caveats

3.3 Validation

3.3.1 Unit tests

3.3.2 Larger-scale performance tests
CHAPTER
FOUR

APPLICATIONS

Placeholder chapter
CHAPTER
FIVE

BRIDGE NETDEVICE

Placeholder chapter
Some examples of the use of Bridge NetDevice can be found in examples/csma/ directory.
Click is a software architecture for building configurable routers. By using different combinations of packet processing units called elements, a Click router can be made to perform a specific kind of functionality. This flexibility provides a good platform for testing and experimenting with different protocols.

### 6.1 Model Description

The source code for the Click model lives in the directory `src/click`.

#### 6.1.1 Design

ns-3’s design is well suited for an integration with Click due to the following reasons:

- Packets in ns-3 are serialised/deserialised as they move up/down the stack. This allows ns-3 packets to be passed to and from Click as they are.
- This also means that any kind of ns-3 traffic generator and transport should work easily on top of Click.
- By striving to implement click as an Ipv4RoutingProtocol instance, we can avoid significant changes to the LL and MAC layer of the ns-3 code.

The design goal was to make the ns-3-click public API simple enough such that the user needs to merely add an Ipv4ClickRouting instance to the node, and inform each Click node of the Click configuration file (.click file) that it is to use.

This model implements the interface to the Click Modular Router and provides the Ipv4ClickRouting class to allow a node to use Click for external routing. Unlike normal Ipv4RoutingProtocol sub types, Ipv4ClickRouting doesn’t use a `RouteInput()` method, but instead, receives a packet on the appropriate interface and processes it accordingly. Note that you need to have a routing table type element in your Click graph to use Click for external routing. This is needed by the `RouteOutput()` function inherited from Ipv4RoutingProtocol. Furthermore, a Click based node uses a different kind of L3 in the form of Ipv4L3ClickProtocol, which is a trimmed down version of Ipv4L3Protocol. Ipv4L3ClickProtocol passes on packets passing through the stack to Ipv4ClickRouting for processing.

**Developing a Simulator API to allow ns-3 to interact with Click**

Much of the API is already well defined, which allows Click to probe for information from the simulator (like a Node’s ID, an Interface ID and so forth). By retaining most of the methods, it should be possible to write new implementations specific to ns-3 for the same functionality.
Hence, for the Click integration with ns-3, a class named Ipv4ClickRouting will handle the interaction with Click. The code for the same can be found in `src/click/model/ipv4-click-routing.{cc,h}.

**Packet hand off between ns-3 and Click**

There are four kinds of packet hand-offs that can occur between ns-3 and Click.

- L4 to L3
- L3 to L4
- L3 to L2
- L2 to L3

To overcome this, we implement Ipv4L3ClickProtocol, a stripped down version of Ipv4L3Protocol. Ipv4L3ClickProtocol passes packets to and from Ipv4ClickRouting appropriately to perform routing.

### 6.1.2 Scope and Limitations

- In its current state, the NS-3 Click Integration is limited to use only with L3, leaving NS-3 to handle L2. We are currently working on adding Click MAC support as well. See the usage section to make sure that you design your Click graphs accordingly.

- Furthermore, ns-3-click will work only with userlevel elements. The complete list of elements are available at [http://read.cs.ucla.edu/click/elements](http://read.cs.ucla.edu/click/elements). Elements that have ‘all’, ‘userlevel’ or ‘ns’ mentioned beside them may be used.

- As of now, the ns-3 interface to Click is Ipv4 only. We will be adding Ipv6 support in the future.

### 6.1.3 References


### 6.2 Usage

#### 6.2.1 Building Click

The first step is to fetch ([http://read.cs.ucla.edu/click/download](http://read.cs.ucla.edu/click/download)) and build Click. At the top of your Click source directory:

```
$: ./configure --enable-userlevel --disable-linuxmodule --enable-nsclick --enable-wifi
$: make
```
The –enable-wifi flag may be skipped if you don’t intend on using Click with Wifi. * Note: You don’t need to do a ‘make install’.

Once Click has been built successfully, change into the ns-3 directory and configure ns-3 with Click Integration support:

$: ./waf configure --enable-examples --enable-tests --with-nsclick=/path/to/click/source

Hint: If you have click installed one directory above ns-3 (such as in the ns-3-allinone directory), and the name of the directory is ‘click’ (or a symbolic link to the directory is named ‘click’), then the –with-nsclick specifier is not necessary; the ns-3 build system will successfully find the directory.

If it says ‘enabled’ beside ‘NS-3 Click Integration Support’, then you’re good to go. Note: If running modular ns-3, the minimum set of modules required to run all ns-3-click examples is wifi, csma and config-store.

Next, try running one of the examples:

$: ./waf --run nsclick-simple-lan

You may then view the resulting .pcap traces, which are named nsclick-simple-lan-0-0.pcap and nsclick-simple-lan-0-1.pcap.

### 6.2.2 Click Graph Instructions

The following should be kept in mind when making your Click graph:

- Only userlevel elements can be used.
- You will need to replace FromDevice and ToDevice elements with FromSimDevice and ToSimDevice elements.
- Packets to the kernel are sent up using ToSimDevice(tap0,IP).
- For any node, the device which sends/receives packets to/from the kernel, is named ‘tap0’. The remaining interfaces should be named eth0, eth1 and so forth (even if you’re using wifi). Please note that the device numbering should begin from 0. In future, this will be made flexible so that users can name devices in their Click file as they wish.
- A routing table element is a mandatory. The OUTports of the routing table element should correspond to the interface number of the device through which the packet will ultimately be sent out. Violating this rule will lead to really weird packet traces. This routing table element’s name should then be passed to the Ipv4ClickRouting protocol object as a simulation parameter. See the Click examples for details.
- The current implementation leaves Click with mainly L3 functionality, with ns-3 handling L2. We will soon begin working to support the use of MAC protocols on Click as well. This means that as of now, Click’s Wifi specific elements cannot be used with ns-3.

### 6.2.3 Debugging Packet Flows from Click

From any point within a Click graph, you may use the Print (http://read.cs.ucla.edu/click/elements/print) element and its variants for pretty printing of packet contents. Furthermore, you may generate pcap traces of packets flowing through a Click graph by using the ToDump (http://read.cs.ucla.edu/click/elements/todump) element as well. For instance:

```
myarpquerier
  -> Print(fromarpquery,64)
  -> ToDump(out_arpquery,PER_NODE 1)
  -> ethout;
```
and ...will print the contents of packets that flow out of the ArpQuerier, then generate a pcap trace file which will have a suffix ‘out_arpquery’, for each node using the Click file, before pushing packets onto ‘ethout’.

6.2.4 Helper

To have a node run Click, the easiest way would be to use the ClickInternetStackHelper class in your simulation script. For instance:

```cpp
ClickInternetStackHelper click;
click.SetClickFile (myNodeContainer, "nsclick-simple-lan.click");
click.SetRoutingTableElement (myNodeContainer, "u/rt");
click.Install (myNodeContainer);
```

The example scripts inside `src/click/examples/` demonstrate the use of Click based nodes in different scenarios. The helper source can be found inside `src/click/helper/click-internet-stack-helper.{h,cc}

6.2.5 Examples

The following examples have been written, which can be found in `src/click/examples/`:

- `nsclick-simple-lan.cc` and `nsclick-raw-wlan.cc`: A Click based node communicating with a normal ns-3 node without Click, using Csma and Wifi respectively. It also demonstrates the use of TCP on top of Click, something which the original nsclik implementation for NS-2 couldn’t achieve.
- `nsclick-udp-client-server-csma.cc` and `nsclick-udp-client-server-wifi.cc`: A 3 node LAN (Csma and Wifi respectively) wherein 2 Click based nodes run a UDP client, that sends packets to a third Click based node running a UDP server.
- `nsclick-routing.cc`: One Click based node communicates to another via a third node that acts as an IP router (using the IP router Click configuration). This demonstrates routing using Click.

Scripts are available within `<click-dir>/conf/` that allow you to generate Click files for some common scenarios. The IP Router used in `nsclick-routing.cc` was generated from the make-ip-conf.pl file and slightly adapted to work with ns-3-click.

6.3 Validation

This model has been tested as follows:

- Unit tests have been written to verify the internals of Ipv4ClickRouting. This can be found in `src/click/ipv4-click-routing-test.cc`. These tests verify whether the methods inside Ipv4ClickRouting which deal with Device name to ID, IP Address from device name and Mac Address from device name bindings work as expected.
- The examples have been used to test Click with actual simulation scenarios. These can be found in `src/click/examples/`. These tests cover the following: the use of different kinds of transports on top of Click, TCP/UDP, whether Click nodes can communicate with non-Click based nodes, whether Click nodes can communicate with each other, using Click to route packets using static routing.
- Click has been tested with Csma, Wifi and Point-to-Point devices. Usage instructions are available in the preceding section.
This is the introduction to CSMA NetDevice chapter, to complement the Csma model doxygen.

7.1 Overview of the CSMA 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.”

7.1.1 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 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 copies 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.

### 7.2 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
The single “speed of light” delay then corresponds to the time it takes for: 1) a signal 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 the following Attributes:

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

### 7.3 CSMA Net Device Model

The CSMA network device appears somewhat like an Ethernet device. The CsmaNetDevice provides the 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 transmit 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 queuing 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 Ethernet-Trailer 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 \(2^{\text{retries}} - 1\) microseconds before a retry is attempted. The default maximum number of retries is 1000.

### 7.4 Using the CsmaNetDevice

The CSMA net devices and channels are typically created and configured using the associated CsmaHelper object. The various ns-3 device helpers generally work in a similar 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:

```cpp
NodeContainer csmaNodes;
```

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

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

```cpp
CsmaHelper csma;
```

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

```cpp
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:

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

We recommend thinking carefully about changing these Attributes, since it can result in behavior that surprises users. We allow this because we believe flexibility is important. As an example of a possibly surprising effect of changing Attributes, consider the following:

The Mtu Attribute indicates the Maximum Transmission Unit to the device. This is the size of the largest Protocol Data Unit (PDU) that the device can send. This Attribute defaults to 1500 bytes and corresponds to a number found in RFC 894, “A Standard for the Transmission of IP Datagrams over Ethernet Networks.” The number is actually derived from the maximum packet size for 10Base5 (full-spec Ethernet) networks – 1518 bytes. If you subtract DIX encapsulation overhead for Ethernet packets (18 bytes) you will end up with a maximum possible data size (MTU) of 1500 bytes. One can also find that the MTU for IEEE 802.3 networks is 1492 bytes. This is because LLC/SNAP encapsulation adds an extra eight bytes of overhead to the packet. In both cases, the underlying network hardware is limited to 1518 bytes, but the MTU is different because the encapsulation is different.

If one leaves the Mtu Attribute at 1500 bytes and changes the encapsulation mode Attribute to Llc, the result will be a network that encapsulates 1500 byte PDUs with LLC/SNAP framing resulting in packets of 1526 bytes. This would be illegal in many networks, but we allow you do do this. This results in a simulation that quite subtly does not reflect what you might be expecting since a real device would balk at sending a 1526 byte packet.

There also exist jumbo frames (1500 < MTU <= 9000 bytes) and super-jumbo (MTU > 9000 bytes) frames that are not officially sanctioned by IEEE but are available in some high-speed (Gigabit) networks and NICs. In the CSMA model, one could leave the encapsulation mode set to Dix, and set the Mtu to 64000 bytes – even though an associated CsmaChannel DataRate was left at 10 megabits per second (certainly not Gigabit Ethernet). This would essentially model an Ethernet switch made out of vampire-tapped 1980s-style 10Base5 networks that support super-jumbo datagrams, which is certainly not something that was ever made, nor is likely to ever be made; however it is quite easy for you to configure.
Be careful about assumptions regarding what CSMA is actually modelling and how configuration (Attributes) may allow you to swerve considerably away from reality.

7.5 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.

7.5.1 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 transmitted 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 definition, fired when a packet cannot be enqued on the transmit queue because 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.”

7.5.2 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.

### 7.6 Summary

The ns3 CSMA model is a simplistic model of an Ethernet-like network. It supports a Carrier-Sense function and allows for Multiple Access to a shared medium. It is not physical in the sense that the state of the medium is instantaneously shared among all devices. This means that there is no collision detection required in this model and none is implemented. There will never be a “jam” of a packet already on the medium. Access to the shared channel is on a first-come first-served basis as determined by the simulator scheduler. If the channel is determined to be busy by looking at the global state, a random exponential backoff is performed and a retry is attempted.

Ns-3 Attributes provide a mechanism for setting various parameters in the device and channel such as addresses, encapsulation modes and error model selection. Trace hooks are provided in the usual manner with a set of upper level hooks corresponding to a transmit queue and used in ASCII tracing; and also a set of lower level hooks used in pcap tracing.

Although the ns-3 CsmaChannel and CsmaNetDevice does not model any kind of network you could build or buy, it does provide us with some useful functionality. You should, however, understand that it is explicitly not Ethernet or any flavor of IEEE 802.3 but an interesting subset.
DSDV ROUTING

Destination-Sequenced Distance Vector (DSDV) routing protocol is a pro-active, table-driven routing protocol for MANETs developed by Charles E. Perkins and Pravin Bhagwat in 1994. It uses the hop count as metric in route selection.

This model was developed by the ResiliNets research group at the University of Kansas. A paper on this model exists at this URL.

8.1 DSDV Routing Overview

DSDV Routing Table: Every node will maintain a table listing all the other nodes it has known either directly or through some neighbors. Every node has a single entry in the routing table. The entry will have information about the node’s IP address, last known sequence number and the hop count to reach that node. Along with these details the table also keeps track of the nexthop neighbor to reach the destination node, the timestamp of the last update received for that node.

The DSDV update message consists of three fields, Destination Address, Sequence Number and Hop Count.

Each node uses 2 mechanisms to send out the DSDV updates. They are,

1. **Periodic Updates** Periodic updates are sent out after every m_periodicUpdateInterval(default:15s). In this update the node broadcasts out its entire routing table.

2. **Trigger Updates** Trigger Updates are small updates in-between the periodic updates. These updates are sent out whenever a node receives a DSDV packet that caused a change in its routing table. The original paper did not clearly mention when for what change in the table should a DSDV update be sent out. The current implementation sends out an update irrespective of the change in the routing table.

The updates are accepted based on the metric for a particular node. The first factor determining the acceptance of an update is the sequence number. It has to accept the update if the sequence number of the update message is higher irrespective of the metric. If the update with same sequence number is received, then the update with least metric (hopCount) is given precedence.

In highly mobile scenarios, there is a high chance of route fluctuations, thus we have the concept of weighted settling time where an update with change in metric will not be advertised to neighbors. The node waits for the settling time to make sure that it did not receive the update from its old neighbor before sending out that update.

The current implementation covers all the above features of DSDV. The current implementation also has a request queue to buffer packets that have no routes to destination. The default is set to buffer up to 5 packets per destination.
8.2 References

Link to the Paper: http://portal.acm.org/citation.cfm?doid=190314.190336
EMULATION OVERVIEW

\( ns-3 \) 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 \texttt{Emu NetDevice} allows \( ns-3 \) simulations to send data on a “real” network. The second kind, called a \texttt{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 \texttt{Emu NetDevice} 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:

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{testbed_emulation.png}
\caption{Example Implementation of Testbed Emulation.}
\end{figure}

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, logging, 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 machine, and also enables virtualization. A simulation of this kind is shown in the following figure:

![Figure 9.2: Implementation overview of emulated channel.](image)

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.

### 9.1 Emu NetDevice

#### 9.1.1 Behavior

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.

One can 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 IPv4 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.

### 9.1.2 Usage

Any mixing of **ns-3** objects with real objects will typically require that **ns-3** compute checksums in its protocols. By default, checksums are not computed by **ns-3**. To enable checksums (e.g. UDP, TCP, IP), users must set the attribute `ChecksumEnabled` to true, such as follows:

```
GlobalValue::Bind ("ChecksumEnabled", BooleanValue (true));
```

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.

- [How to use VMware to set up virtual networks (Windows)](http://www.nsnam.org/wiki/index.php/HOWTO_use_VMware_to_set_up_virtual_networks_(Windows))
- [How to use ns-3 scripts to drive real hardware (experimental)](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
// have assigned to our devices are unique. Ns-3 will happily
// automatically assign the same MAC address 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 server is 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");

For examples that use the Emu net device, see src/emu/examples/emu-udp-echo.cc and src/emu/examples/emu-ping.cc in the repository http://code.nsnam.org/craigdo/ns-3-emu/.

9.1.3 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 \texttt{Emu} and the \texttt{Tap} devices. The high-level view is that the \texttt{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.

The \texttt{Emu} net device uses the \texttt{ns-3} threading and multithreaded real-time scheduler extensions. The interesting work in the \texttt{Emu} device is done when the net device is started (\texttt{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 \texttt{m\_deviceName}:

\begin{verbatim}
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));
\end{verbatim}

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 \texttt{Up}:

\begin{verbatim}
m\_readThread = Create\textlt{SystemThread} ( MakeCallback (&EmuNetDevice::ReadThread, this));
m\_readThread->Start ();

NotifyLinkUp ();
\end{verbatim}

The \texttt{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.

\begin{verbatim}
for (;;) {
  ...
  len = recvfrom (m\_sock, buf, bufferSize, 0, (struct sockaddr *)\&addr, 
                  \&addrSize);
  ...
  DynamicCast<RealtimeSimulatorImpl> (Simulator::GetImplementation ())->
    ScheduleRealtimeNow ( MakeEvent (&EmuNetDevice::ForwardUp, this, buf, len));
  ...
}
\end{verbatim}

The line starting with our templated DynamicCast function probably deserves a comment. It gains access to the simulator implementation object using the \texttt{Simulator::GetImplementation} method and then casts to the

\section{9.1. Emu NetDevice}
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:

```cpp
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.

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 `Emu` net device for transmission it always passes through the transmit queue. The transmit queue in the `EmuNetDevice` 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 `EmuNetDevice` 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::EmuNetDevice::Send` or `ns3::EmuNetDevice::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 happen immediately after the packet was enqueued and only indicate that the packet is about to be sent to an underlying raw socket. The actual time at which the packet is sent out on the wire is not available.

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 underlying raw socket.

The trace source m_dropTrace is not used in the Emu net device since that is really the business of the underlying “real” device driver.

The other low-level trace source fires on reception of an accepted packet (see ns3::EmuNetDevice::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 Emu net device.

9.2 Tap NetDevice

The Tap NetDevice can be used to allow a host system or virtual machines to interact with a simulation.

9.2.1 TapBridge Model Overview

The Tap Bridge is designed to integrate “real” internet hosts (or more precisely, hosts that support Tun/Tap devices) into ns-3 simulations. The goal is to make it appear to a “real” host node in that it has an ns-3 net device as a local device. The concept of a “real host” is a bit slippery since the “real host” may actually be virtualized using readily available technologies such as VMware, VirtualBox or OpenVZ.

Since we are, in essence, connecting the inputs and outputs of an ns-3 net device to the inputs and outputs of a Linux Tap net device, we call this arrangement a Tap Bridge.

There are three basic operating modes of this device available to users. Basic functionality is essentially identical, but the modes are different in details regarding how the arrangement is created and configured; and what devices can live on which side of the bridge.

We call these three modes the ConfigureLocal, UseLocal and UseBridge modes. The first “word” in the camel case mode identifier indicates who has the responsibility for creating and configuring the taps. For example, the “Configure” in ConfigureLocal mode indicates that it is the TapBridge that has responsibility for configuring the tap. In UseLocal mode and UseBridge modes, the “Use” prefix indicates that the TapBridge is asked to “Use” an existing configuration.

In other words, in ConfigureLocal mode, the TapBridge has the responsibility for creating and configuring the TAP devices. In UseBridge or UseLocal modes, the user provides a configuration and the TapBridge adapts to that configuration.

TapBridge ConfigureLocal Mode

In the ConfigureLocal mode, the configuration of the tap device is ns-3 configuration-centric. Configuration information is taken from a device in the ns-3 simulation and a tap device matching the ns-3 attributes is automatically created. In this case, a Linux computer is made to appear as if it was directly connected to a simulated ns-3 network.

This is illustrated below:

```
+--------+
<p>| Linux  |</p>
<table>
<thead>
<tr>
<th>host</th>
</tr>
</thead>
<tbody>
<tr>
<td>apps</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>stack</td>
</tr>
<tr>
<td>------</td>
</tr>
</tbody>
</table>
```

```
+--------+
<p>| ghost  |</p>
<table>
<thead>
<tr>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>stack</td>
</tr>
<tr>
<td>node</td>
</tr>
</tbody>
</table>
```

9.2. Tap NetDevice
In this case, the “ns-3 net device” in the “ghost node” appears as if it were actually replacing the TAP device in the Linux host. The ns-3 simulation creates the TAP device on the underlying Linux OS and configures the IP and MAC addresses of the TAP device to match the values assigned to the simulated ns-3 net device. The “IPC” link shown above is the network tap mechanism in the underlying OS. The whole arrangement acts as a conventional bridge; but a bridge between devices that happen to have the same shared MAC and IP addresses.

Here, the user is not required to provide any configuration information specific to the tap. A tap device will be created and configured by ns-3 according to its defaults, and the tap device will have its name assigned by the underlying operating system according to its defaults.

If the user has a requirement to access the created tap device, he or she may optionally provide a “DeviceName” attribute. In this case, the created OS tap device will be named accordingly.

The ConfigureLocal mode is the default operating mode of the Tap Bridge.

**TapBridge UseLocal Mode**

The UseLocal mode is quite similar to the ConfigureLocal mode. The significant difference is, as the mode name implies, the TapBridge is going to “Use” an existing tap device previously created and configured by the user. This mode is particularly useful when a virtualization scheme automatically creates tap devices and ns-3 is used to provide simulated networks for those devices.

In this case, the pre-configured MAC address of the “Tap device” (MAC X) will not be the same as that of the bridged
“ns-3 net device” (MAC Y) shown in the illustration above. In order to bridge to ns-3 net devices which do not support SendFrom() (especially wireless STA nodes) we impose a requirement that only one Linux device (with one unique MAC address – here X) generates traffic that flows across the IPC link. This is because the MAC addresses of traffic across the IPC link will be “spoofed” or changed to make it appear to Linux and ns-3 that they have the same address. That is, traffic moving from the Linux host to the ns-3 ghost node will have its MAC address changed from X to Y and traffic from the ghost node to the Linux host will have its MAC address changed from Y to X. Since there is a one-to-one correspondence between devices, there may only be one MAC source flowing from the Linux side. This means that Linux bridges with more than one net device added are incompatible with UseLocal mode.

In UseLocal mode, the user is expected to create and configure a tap device completely outside the scope of the ns-3 simulation using something like:

```
sudo tunctl -t tap0
sudo ifconfig tap0 hw ether 08:00:2e:00:00:01
sudo ifconfig tap0 10.1.1.1 netmask 255.255.255.0 up
```

To tell the TapBridge what is going on, the user will set either directly into the TapBridge or via the TapBridgeHelper, the “DeviceName” attribute. In the case of the configuration above, the “DeviceName” attribute would be set to “tap0” and the “Mode” attribute would be set to “UseLocal”.

One particular use case for this mode is in the OpenVZ environment. There it is possible to create a Tap device on the “Hardware Node” and move it into a Virtual Private Server. If the TapBridge is able to use an existing tap device it is then possible to avoid the overhead of an OS bridge in that environment.

**TapBridge UseBridge Mode**

The simplest mode for those familiar with Linux networking is the UseBridge mode. Again, the “Use” prefix indicates that the TapBridge is going to Use an existing configuration. In this case, the TapBridge is going to logically extend a Linux bridge into ns-3.

This is illustrated below:

```
+---------+         +---------+
| Linux |         | ghost |
| ------|         | node |
| apps  |         | stack |
| ------| +--------|-- -------|
| stack |         | IP | stack |
| Virtual| TAP | Device | Device| <---- IPC ------>
| ------| +--------|-- -------|
| ns-3  |         | ns-3 |
| OS (brctl) Bridge |         | device |
+--------+ +--------+ +--------+
| ns-3 channel |
```

In this case, a computer running Linux applications, protocols, etc., is connected to a ns-3 simulated network in such a way as to make it appear to the Linux host that the TAP device is a real network device participating in the Linux bridge.

In the ns-3 simulation, a TapBridge is created to match each TAP Device. The name of the TAP Device is assigned to the Tap Bridge using the “DeviceName” attribute. The TapBridge then logically extends the OS bridge to encompass...
the ns-3 net device.

Since this mode logically extends an OS bridge, there may be many Linux net devices on the non-ns-3 side of the bridge. Therefore, like a net device on any bridge, the ns-3 net device must deal with the possibly of many source addresses. Thus, ns-3 devices must support SendFrom() (NetDevice::SupportsSendFrom() must return true) in order to be configured for use in UseBridge mode.

It is expected that the user will do something like the following to configure the bridge and tap completely outside ns-3:

```
sudo brctl addbr mybridge
sudo tunctl -t mytap
sudo ifconfig mytap hw ether 00:00:00:00:00:01
sudo ifconfig mytap 0.0.0.0 up
sudo brctl addif mybridge mytap
sudo brctl addif mybridge ...
sudo ifconfig mybridge 10.1.1.1 netmask 255.255.255.0 up
```

To tell the TapBridge what is going on, the user will set either directly into the TapBridge or via the TapBridgeHelper, the “DeviceName” attribute. In the case of the configuration above, the “DeviceName” attribute would be set to “mytap” and the “Mode” attribute would be set to “UseBridge”.

This mode is especially useful in the case of virtualization where the configuration of the virtual hosts may be dictated by another system and not be changable to suit ns-3. For example, a particular VM scheme may create virtual “vethx” or “vmnetx” devices that appear local to virtual hosts. In order to connect to such systems, one would need to manually create TAP devices on the host system and bridge these TAP devices to the existing (VM) virtual devices. The job of the Tap Bridge in this case is to extend the bridge to join a ns-3 net device.

**TapBridge ConfigureLocal Operation**

In ConfigureLocal mode, the TapBridge and therefore its associated ns-3 net device appears to the Linux host computer as a network device just like any arbitrary “eth0” or “ath0” might appear. The creation and configuration of the TAP device is done by the ns-3 simulation and no manual configuration is required by the user. The IP addresses, MAC addresses, gateways, etc., for created TAP devices are extracted from the simulation itself by querying the configuration of the ns-3 device and the TapBridge Attributes.

Since the MAC addresses are identical on the Linux side and the ns-3 side, we can use Send() on the ns-3 device which is available on all ns-3 net devices. Since the MAC addresses are identical there is no requirement to hook the promiscuous callback on the receive side. Therefore there are no restrictions on the kinds of net device that are usable in ConfigureLocal mode.

The TapBridge appears to an ns-3 simulation as a channel-less net device. This device must not have an IP address associated with it, but the bridged (ns-3) net device must have an IP address. Be aware that this is the inverse of an ns-3 BridgeNetDevice (or a conventional bridge in general) which demands that its bridge ports not have IP addresses, but allows the bridge device itself to have an IP address.

The host computer will appear in a simulation as a “ghost” node that contains one TapBridge for each NetDevice that is being bridged. From the perspective of a simulation, the only difference between a ghost node and any other node will be the presence of the TapBridge devices. Note however, that the presence of the TapBridge does affect the connectivity of the net device to the IP stack of the ghost node.

Configuration of address information and the ns-3 devices is not changed in any way if a TapBridge is present. A TapBridge will pick up the addressing information from the ns-3 net device to which it is connected (its “bridged” net device) and use that information to create and configure the TAP device on the real host.

The end result of this is a situation where one can, for example, use the standard ping utility on a real host to ping a simulated ns-3 node. If correct routes are added to the internet host (this is expected to be done automatically in future
The Tap Bridge lives in a kind of a gray world somewhere between a Linux host and an ns-3 bridge device. From the Linux perspective, this code appears as the user mode handler for a TAP net device. In ConfigureLocal mode, this Tap device is automatically created by the ns-3 simulation. When the Linux host writes to one of these automatically created /dev/tap devices, the write is redirected into the TapBridge that lives in the ns-3 world; and from this perspective, the packet write on Linux becomes a packet read in the Tap Bridge. In other words, a Linux process writes a packet to a tap device and this packet is redirected by the network tap mechanism to an ns-3 process where it is received by the TapBridge as a result of a read operation there. The TapBridge then writes the packet to the ns-3 net device to which it is bridged; and therefore it appears as if the Linux host sent a packet directly through an ns-3 net device onto an ns-3 network.

In the other direction, a packet received by the ns-3 net device connected to the Tap Bridge is sent via a receive callback to the TapBridge. The TapBridge then takes that packet and writes it back to the host using the network tap mechanism. This write to the device will appear to the Linux host as if a packet has arrived on a net device; and therefore as if a packet received by the ns-3 net device during a simulation has appeared on a real Linux net device.

The upshot is that the Tap Bridge appears to bridge a tap device on a Linux host in the “real world” to an ns-3 net device in the simulation. Because the TAP device and the bridged ns-3 net device have the same MAC address and the network tap IPC link is not externalized, this particular kind of bridge makes it appear that a ns-3 net device is actually installed in the Linux host.

In order to implement this on the ns-3 side, we need a “ghost node” in the simulation to hold the bridged ns-3 net device and the TapBridge. This node should not actually do anything else in the simulation since its job is simply to make the net device appear in Linux. This is not just arbitrary policy, it is because:

- Bits sent to the TapBridge from higher layers in the ghost node (using the TapBridge Send method) are completely ignored. The TapBridge is not, itself, connected to any network, neither in Linux nor in ns-3. You can never send nor receive data over a TapBridge from the ghost node.

- The bridged ns-3 net device has its receive callback disconnected from the ns-3 node and reconnected to the Tap Bridge. All data received by a bridged device will then be sent to the Linux host and will not be received by the node. From the perspective of the ghost node, you can send over this device but you cannot ever receive.

Of course, if you understand all of the issues you can take control of your own destiny and do whatever you want – we do not actively prevent you from using the ghost node for anything you decide. You will be able to perform typical ns-3 operations on the ghost node if you so desire. The internet stack, for example, must be there and functional on that node in order to participate in IP address assignment and global routing. However, as mentioned above, interfaces talking to any TapBridge or associated bridged net devices will not work completely. If you understand exactly what you are doing, you can set up other interfaces and devices on the ghost node and use them; or take advantage of the operational send side of the bridged devices to create traffic generators. We generally recommend that you treat this node as a ghost of the Linux host and leave it to itself, though.

### TapBridge UseLocal Mode Operation

As described in above, the TapBridge acts like a bridge from the “real” world into the simulated ns-3 world. In the case of the ConfigureLocal mode, life is easy since the IP address of the Tap device matches the IP address of the ns-3 device and the MAC address of the Tap device matches the MAC address of the ns-3 device; and there is a one-to-one relationship between the devices.

Things are slightly complicated when a Tap device is externally configured with a different MAC address than the ns-3 net device. The conventional way to deal with this kind of difference is to use promiscuous mode in the bridged device to receive packets destined for the different MAC address and forward them off to Linux. In order to move packets the other way, the conventional solution is SendFrom() which allows a caller to “spoof” or change the source MAC address to match the different Linux MAC address.
We do have a specific requirement to be able to bridge Linux Virtual Machines onto wireless STA nodes. Unfortunately, the 802.11 spec doesn’t provide a good way to implement SendFrom(), so we have to work around that problem.

To this end, we provided the UseLocal mode of the Tap Bridge. This mode allows you approach the problem as if you were creating a bridge with a single net device. A single allowed address on the Linux side is remembered in the TapBridge, and all packets coming from the Linux side are repeated out the ns-3 side using the ns-3 device MAC source address. All packets coming in from the ns-3 side are repeated out the Linux side using the remembered MAC address. This allows us to use Send() on the ns-3 device side which is available on all ns-3 net devices.

UseLocal mode is identical to the ConfigureLocal mode except for the creation and configuration of the tap device and the MAC address spoofing.

**TapBridge UseBridge Operation**

As described in the ConfigureLocal mode section, when the Linux host writes to one of the /dev/tap devices, the write is redirected into the TapBridge that lives in the ns-3 world. In the case of the UseBridge mode, these packets will need to be sent out on the ns-3 network as if they were sent on a device participating in the Linux bridge. This means calling the SendFrom() method on the bridged device and providing the source MAC address found in the packet.

In the other direction, a packet received by an ns-3 net device is hooked via callback to the TapBridge. This must be done in promiscuous mode since the goal is to bridge the ns-3 net device onto the OS (brctl) bridge of which the TAP device is a part.

For these reasons, only ns-3 net devices that support SendFrom() and have a hookable promiscuous receive callback are allowed to participate in UseBridge mode TapBridge configurations.

**9.2.2 Tap Bridge Channel Model**

There is no channel model associated with the Tap Bridge. In fact, the intention is make it appear that the real internet host is connected to the channel of the bridged net device.

**9.2.3 Tap Bridge Tracing Model**

Unlike most ns-3 devices, the TapBridge does not provide any standard trace sources. This is because the bridge is an intermediary that is essentially one function call away from the bridged device. We expect that the trace hooks in the bridged device will be sufficient for most users.

**9.2.4 Using the TapBridge**

We expect that most users will interact with the TapBridge device through the TapBridgeHelper. Users of other helper classes, such as CSMA or Wifi, should be comfortable with the idioms used there.
ENERGY FRAMEWORK

Energy consumption is a key issue for wireless devices, and wireless network researchers often need to investigate the energy consumption at a node or in the overall network while running network simulations in ns-3. This requires ns-3 to support energy consumption modeling. Further, as concepts such as fuel cells and energy scavenging are becoming viable for low power wireless devices, incorporating the effect of these emerging technologies into simulations requires support for modeling diverse energy sources in ns-3. The ns-3 Energy Framework provides the basis for energy consumption and energy source modeling.

10.1 Model Description

The source code for the Energy Framework is currently at: src/energy.

10.1.1 Design

The ns-3 Energy Framework is composed of 2 parts: Energy Source and Device Energy Model. The framework will be implemented into the src/energy/models folder.

Energy Source

The Energy Source represents the power supply on each node. A node can have one or more energy sources, and each energy source can be connected to multiple device energy models. Connecting an energy source to a device energy model implies that the corresponding device draws power from the source. The basic functionality of the Energy Source is to provide energy for devices on the node. When energy is completely drained from the Energy Source, it notifies the devices on node such that each device can react to this event. Further, each node can access the Energy Source Objects for information such as remaining energy or energy fraction (battery level). This enables the implementation of energy aware protocols in ns-3.

In order to model a wide range of power supplies such as batteries, the Energy Source must be able to capture characteristics of these supplies. There are 2 important characteristics or effects related to practical batteries:

- Rate Capacity Effect: Decrease of battery lifetime when the current draw is higher than the rated value of the battery.
- Recovery Effect: Increase of battery lifetime when the battery is alternating between discharge and idle states.

In order to incorporate the Rate Capacity Effect, the Energy Source uses current draw from all devices on the same node to calculate energy consumption. The Energy Source polls all devices on the same node periodically to calculate the total current draw and hence the energy consumption. When a device changes state, its corresponding Device Energy Model will notify the Energy Source of this change and new total current draw will be calculated.
The Energy Source base class keeps a list of devices (Device Energy Model objects) using the particular Energy Source as power supply. When energy is completely drained, the Energy Source will notify all devices on this list. Each device can then handle this event independently, based on the desired behavior when power supply is drained.

**Device Energy Model**

The Device Energy Model is the energy consumption model of a device on node. It is designed to be a state based model where each device is assumed to have a number of states, and each state is associated with a power consumption value. Whenever the state of the device changes, the corresponding Device Energy Model will notify the Energy Source of the new current draw of the device. The Energy Source will then calculate the new total current draw and update the remaining energy.

The Device Energy Model can also be used for devices that do not have finite number of states. For example, in an electric vehicle, the current draw of the motor is determined by its speed. Since the vehicle’s speed can take continuous values within a certain range, it is infeasible to define a set of discrete states of operation. However, by converting the speed value into current directly, the same set of Device Energy Model APIs can still be used.

**10.1.2 Future Work**

For Device Energy Models, we are planning to include support for other PHY layer models provided in ns-3 such as WiMAX. For Energy Sources, we are planning to included new types of Energy Sources such as energy scavenging.

**10.1.3 References**

**10.2 Usage**

The main way that ns-3 users will typically interact with the Energy Framework is through the helper API and through the publicly visible attributes of the framework. The helper API is defined in `src/energy/helper/*.h`.

In order to use the energy framework, the user must install an Energy Source for the node of interest and the corresponding Device Energy Model for the network devices. Energy Source (objects) are aggregated onto each node by the Energy Source Helper. In order to allow multiple energy sources per node, we aggregate an Energy Source Container rather than directly aggregating a source object.

The Energy Source object also keeps a list of Device Energy Model objects using the source as power supply. Device Energy Model objects are installed onto the Energy Source by the Device Energy Model Helper. User can access the Device Energy Model objects through the Energy Source object to obtain energy consumption information of individual devices.

**10.2.1 Examples**

The example directories, `src/examples/energy` and `examples/energy`, contain some basic code that shows how to set up the framework.

**10.2.2 Helpers**

**Energy Source Helper**

Base helper class for Energy Source objects, this helper Aggregates Energy Source object onto a node. Child implementation of this class creates the actual Energy Source object.
Device Energy Model Helper

Base helper class for Device Energy Model objects, this helper attaches Device Energy Model objects onto Energy Source objects. Child implementation of this class creates the actual Device Energy Model object.

10.2.3 Attributes

Attributes differ between Energy Sources and Devices Energy Models implementations, please look at the specific child class for details.

Basic Energy Source

- BasicEnergySourceInitialEnergyJ: Initial energy stored in basic energy source.
- BasicEnergySupplyVoltageV: Initial supply voltage for basic energy source.
- PeriodicEnergyUpdateInterval: Time between two consecutive periodic energy updates.

RV Battery Model

- RvBatteryModelPeriodicEnergyUpdateInterval: RV battery model sampling interval.
- RvBatteryModelOpenCircuitVoltage: RV battery model open circuit voltage.
- RvBatteryModelCutoffVoltage: RV battery model cutoff voltage.
- RvBatteryModelAlphaValue: RV battery model alpha value.
- RvBatteryModelBetaValue: RV battery model beta value.
- RvBatteryModelNumOfTerms: The number of terms of the infinite sum for estimating battery level.

WiFi Radio Energy Model

- IdleCurrentA: The default radio Idle current in Ampere.
- CcaBusyCurrentA: The default radio CCA Busy State current in Ampere.
- TxCurrentA: The radio Tx current in Ampere.
- RxCurrentA: The radio Rx current in Ampere.
- SwitchingCurrentA: The default radio Channel Switch current in Ampere.

10.2.4 Tracing

Traced values differ between Energy Sources and Devices Energy Models implementations, please look at the specific child class for details.

Basic Energy Source

- RemainingEnergy: Remaining energy at BasicEnergySource.
RV Battery Model

- `RvBatteryModelBatteryLevel`: RV battery model battery level.
- `RvBatteryModelBatteryLifetime`: RV battery model battery lifetime.

WiFi Radio Energy Model

- `TotalEnergyConsumption`: Total energy consumption of the radio device.

10.2.5 Validation

Comparison of the Energy Framework against actual devices have not been performed. Current implementation of the Energy Framework is checked numerically for computation errors. The RV battery model is validated by comparing results with what was presented in the original RV battery model paper.
Placeholder chapter

This feature was added as contributed code (src/contrib) in ns-3.6 and to the main distribution (src/flow-monitor) for ns-3.7. A paper on this feature is published in the proceedings of NSTools: http://www.nstools.org/techprog.shtml.
12.1 Internet Stack

12.1.1 Internet stack aggregation

A bare 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 provides implementation of TCP/IPv4- and IPv6-related components. These include IPv4, ARP, UDP, TCP, IPv6, Neighbor Discovery, 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 InternetStackHelper::Install() which does the following to all nodes passed in as arguments::

```cpp
void
InternetStackHelper::Install (Ptr<Node> node) const
{
  if (node->GetObject<Ipv4> () != 0)
  {
    NS_FATAL_ERROR ("InternetStackHelper::Install(): Aggregating 
                      " "an InternetStack to a node with an existing Ipv4 object");
    return;
  }

  CreateAndAggregateObjectFromTypeId (node, "ns3::ArpL3Protocol");
  CreateAndAggregateObjectFromTypeId (node, "ns3::Ipv4L3Protocol");
  CreateAndAggregateObjectFromTypeId (node, "ns3::Icmpv4L4Protocol");
  CreateAndAggregateObjectFromTypeId (node, "ns3::UdpL4Protocol");
  node->AggregateObject (m_tcpFactory.Create<Object>());
  Ptr<PacketSocketFactory> factory = CreateObject<PacketSocketFactory> ();
  node->AggregateObject (factory);
  // Set routing
  Ptr<Ipv4> ipv4 = node->GetObject<Ipv4> ();
  Ptr<Ipv4RoutingProtocol> ipv4Routing = m_routing->Create (node);  
  ipv4->SetRoutingProtocol (ipv4Routing);
}
```

Where multiple implementations exist in ns-3 (TCP, IP routing), these objects are added by a factory object (TCP) or by a routing helper (m_routing).

Note that the routing protocol is configured and set outside this function. By default, the following protocols are added to Ipv4::
InternetStackHelper::InternetStackHelper ()
{
    SetTcp ("ns3::TcpL4Protocol");
    static Ipv4StaticRoutingHelper staticRouting;
    static Ipv4GlobalRoutingHelper globalRouting;
    static Ipv4ListRoutingHelper listRouting;
    listRouting.Add (staticRouting, 0);
    listRouting.Add (globalRouting, -10);
    SetRoutingHelper (listRouting);
}

By default, IPv4 and IPv6 are enabled.

Internet Node structure

An IPv4-capable Node (an ns-3 Node augmented by aggregation to have one or more IP stacks) has the following internal structure.

Layer-3 protocols

At the lowest layer, sitting above the NetDevices, are the “layer 3” protocols, including IPv4, IPv6 (in the future), and ARP. The class Ipv4L3Protocol is an implementation class whose public interface is typically class Ipv4, but the Ipv4L3Protocol public API is also used internally at present.

In class Ipv4L3Protocol, one method described below is Receive ()::

    void Receive( Ptr<NetDevice> device, Ptr<const Packet> p, uint16_t protocol,
                 const Address &from, const Address &to, NetDevice::PacketType packetType);;

First, note that the Receive () function has a matching signature to the ReceiveCallback in the class Node. This function pointer is inserted into 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 routing has determined that a packet is for the local node, it forwards it up the stack. This is done with the following function:

```c
void
Ipv4L3Protocol::LocalDeliver (Ptr<const Packet> packet, Ipv4Header const&ip, uint32_t iif)
```

The first step is to find the right Ipv4L4Protocol object, 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.

The IPv6 implementation follows a similar architecture.

### 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:

```c
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:

```c
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`:

```c
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.

Ipv4-capable node interfaces

Many of the implementation details, or internal objects themselves, of Ipv4-capable 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/network directory, including principally:
• address.h
• socket.h
• node.h
• packet.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.

Example path of a packet

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

![Diagram of packet flow]

Figure 12.1: Send path of a packet.

12.2 IPv4

Placeholder chapter
12.3 IPv6

Placeholder chapter

IPv6 models are being added to ns-3. A paper on the IPv6 models was published in WNS2 2008: http://lsiit.u-strasbg.fr/Publications/2008/VMM08/.

12.4 Routing overview

ns-3 is intended to support traditional routing approaches and protocols, support ports of open source routing implementations, and facilitate research into unorthodox routing techniques. The overall routing architecture is described below in Routing architecture. Users who wish to just read about how to configure global routing for wired topologies can read Global centralized routing. Unicast routing protocols are described in Unicast routing. Multicast routing is documented in Multicast routing.

12.4.1 Routing architecture

Overview of routing shows the overall routing architecture for Ipv4. The key objects are Ipv4L3Protocol, Ipv4RoutingProtocol(s) (a class to which all routing/forwarding has been delegated from Ipv4L3Protocol), and Ipv4Route(s).

Ipv4L3Protocol must have at least one Ipv4RoutingProtocol added to it at simulation setup time. This is done explicitly by calling Ipv4::SetRoutingProtocol().

The abstract base class Ipv4RoutingProtocol() declares a minimal interface, consisting of two methods: RouteOutput() and RouteInput(). For packets traveling outbound from a host, the transport protocol will query Ipv4 for the Ipv4RoutingProtocol object interface, and will request a route via Ipv4RoutingProtocol::RouteOutput(). A Ptr to
The Ipv4Route object is returned. This is analogous to a dst_cache entry in Linux. The Ipv4Route is carried down to the Ipv4L3Protocol to avoid a second lookup there. However, some cases (e.g. Ipv4 raw sockets) will require a call to RouteOutput() directly from Ipv4L3Protocol.

For packets received inbound for forwarding or delivery, the following steps occur. Ipv4L3Protocol::Receive() calls Ipv4RoutingProtocol::RouteInput(). This passes the packet ownership to the Ipv4RoutingProtocol object. There are four callbacks associated with this call:

- LocalDeliver
- UnicastForward
- MulticastForward
- Error

The Ipv4RoutingProtocol must eventually call one of these callbacks for each packet that it takes responsibility for. This is basically how the input routing process works in Linux.

This overall architecture is designed to support different routing approaches, including (in the future) a Linux-like policy-based routing implementation, proactive and on-demand routing protocols, and simple routing protocols for when the simulation user does not really care about routing.

**Ipv4Routing specialization.** illustrates how multiple routing protocols derive from this base class. A class Ipv4ListRouting (implementation class Ipv4ListRoutingImpl) provides the existing list routing approach in ns-3. Its API is the same as base class Ipv4Routing except for the ability to add multiple prioritized routing protocols (Ipv4ListRouting::AddRoutingProtocol(), Ipv4ListRouting::GetRoutingProtocol()).

The details of these routing protocols are described below in Unicast routing. For now, we will first start with a basic unicast routing capability that is intended to globally build routing tables at simulation time t=0 for simulation users who do not care about dynamic routing.
12.4.2 Global centralized routing

Global centralized routing is sometimes called “God” routing; it is a special implementation that walks the simulation topology and runs a shortest path algorithm, and populates each node’s routing tables. No actual protocol overhead (on the simulated links) is incurred with this approach. It does have a few constraints:

- **Wired only**: It is not intended for use in wireless networks.
- **Unicast only**: It does not do multicast.
- **Scalability**: Some users of this on large topologies (e.g., 1000 nodes) have noticed that the current implementation is not very scalable. The global centralized routing will be modified in the future to reduce computations and runtime performance.

Presently, global centralized IPv4 unicast routing over both point-to-point and shared (CSMA) links is supported. By default, when using the ns-3 helper API and the default InternetStackHelper, global routing capability will be added to the node, and global routing will be inserted as a routing protocol with lower priority than the static routes (i.e., users can insert routes via Ipv4StaticRouting API and they will take precedence over routes found by global routing).

### Global Unicast Routing API

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

```cpp
#include "ns3/internet-module.h"
```

If the default InternetStackHelper is used, then an instance of global routing will be aggregated to each node. 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:

```cpp
Ipv4GlobalRoutingHelper::PopulateRoutingTables();
```

**Note**: A reminder that the wifi NetDevice will work but does not take any wireless effects into account. For wireless, we recommend OLSR dynamic routing described below.

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

```cpp
Ipv4GlobalRoutingHelper::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),
   &Ipv4GlobalRoutingHelper::RecomputeRoutingTables);

There are two attributes that govern the behavior. The first is Ipv4GlobalRouting::RandomEcmpRouting. If set to true, packets are randomly routed across equal-cost multipath routes. If set to false (default), only one route is consistently used. The second is Ipv4GlobalRouting::RespondToInterfaceEvents. If set to true, dynamically recompute the global routes upon Interface notification events (up/down, or add/remove address). If set to false (default), routing may break unless the user manually calls RecomputeRoutingTables() after such events. The default is set to false to preserve legacy ns-3 program behavior.

Global Routing Implementation

This section is for those readers who care about how this is implemented. 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.

It is important to note that all of these computations are done before packets are flowing in the network. In particular, there are no overhead or control packets being exchanged when using this implementation. Instead, this global route manager just walks the list of nodes to build the necessary information and configure each node’s routing table.

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. Equal-cost multipath is also supported. Although wireless link types are supported by the implementation, note that due to the nature of this implementation, any channel effects will not be considered and the routing tables will assume that every node on the same shared channel is reachable from every other node (i.e. it will be treated like a broadcast CSMA link).

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.

### 12.4.3 Unicast routing

There are presently seven unicast routing protocols defined for IPv4 and two for IPv6:

- class Ipv4StaticRouting (covering both unicast and multicast)
- IPv4 Optimized Link State Routing (OLSR) (a MANET protocol defined in RFC 3626)
- IPv4 Ad Hoc On Demand Distance Vector (AODV) (a MANET protocol defined in RFC 3561)
- IPv4 Destination Sequenced Distance Vector (DSDV) (a MANET protocol)
- class Ipv4ListRouting (used to store a prioritized list of routing protocols)
- class Ipv4GlobalRouting (used to store routes computed by the global route manager, if that is used)
- class Ipv4NixVectorRouting (a more efficient version of global routing that stores source routes in a packet header field)
- class Ipv6ListRouting (used to store a prioritized list of routing protocols)
- class Ipv6StaticRouting

In the future, this architecture should also allow someone to implement a Linux-like implementation with routing cache, or a Click modular router, but those are out of scope for now.

**Ipv4ListRouting**

This section describes the current default *ns-3* Ipv4RoutingProtocol. 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 facilitate 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.

**Ipv4ListRouting::AddRoutingProtocol**

Class Ipv4ListRouting provides a pure virtual function declaration for the method that allows one to add a routing protocol:

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

This method is implemented by class Ipv4ListRoutingImpl 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. By default in *ns-3*, the helper classes will instantiate a Ipv4ListRoutingImpl object, and add to it an Ipv4StaticRoutingImpl object 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.:
listRoutingPtr->AddRoutingProtocol(myRoutingProto, -10);

Upon calls to RouteOutput() or RouteInput(), the list routing object will search the list of routing protocols, in priority order, until a route is found. Such routing protocol will invoke the appropriate callback and no further routing protocols will be searched.

**Optimized Link State Routing (OLSR)**

This IPv4 routing protocol was originally ported from the OLSR-UM implementation for ns-2. The implementation is found in the src/olsr directory, and an example script is in examples/simple-point-to-point-olsr.cc.

Typically, OLSR is enabled in a main program by use of an OlsrHelper class that installs OLSR into an Ipv4ListRoutingProtocol object. The following sample commands will enable OLSR in a simulation using this helper class along with some other routing helper objects. The setting of priority value 10, ahead of the staticRouting priority of 0, means that OLSR will be consulted for a route before the node’s static routing table.:

```cpp
NodeContainer c:
...
// Enable OLSR
NS_LOG_INFO("Enabling OLSR Routing.");
OlsrHelper olsr;

Ipv4StaticRoutingHelper staticRouting;
Ipv4ListRoutingHelper list;
list.Add (staticRouting, 0);
list.Add (olsr, 10);

InternetStackHelper internet;
iinternet.SetRoutingHelper (list);
iinternet.Install (c);
```

Once installed, the OLSR “main interface” can be set with the SetMainInterface() command. If the user does not specify a main address, the protocol will select the first primary IP address that it finds, starting first the loopback interface and then the next non-loopback interface found, in order of Ipv4 interface index. The loopback address of 127.0.0.1 is not selected. In addition, a number of protocol constants are defined in olsr-routing-protocol.cc.

Olsr is started at time zero of the simulation, based on a call to Object::Start() that eventually calls OlsrRoutingProtocol::DoStart(). Note: a patch to allow the user to start and stop the protocol at other times would be welcome.

Presently, OLSR is limited to use with an Ipv4ListRouting object, and does not respond to dynamic changes to a device’s IP address or link up/down notifications; i.e. the topology changes are due to loss/gain of connectivity over a wireless channel.

### 12.4.4 Multicast routing

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

```cpp
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 explicitly 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:

```cpp
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:

```cpp
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);
```

**12.5 TCP models in ns-3**

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

**12.5.1 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/network 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/internet/model/tcp-socket.{cc,h}. This class exists for hosting TcpSocket attributes that can be reused across different implementations. For instance, the attribute InitialCwnd can be used for any of the implementations that derive from class TcpSocket.

• class TcpSocketFactory: This is used by the layer-4 protocol instance to create TCP sockets of the right type.

There are presently two implementations of TCP available for ns-3.

• a natively implemented TCP for ns-3
• support for the Network Simulation Cradle (NSC)

It should also be mentioned that various ways of combining virtual machines with ns-3 makes available also some additional TCP implementations, but those are out of scope for this chapter.

### 12.5.2 ns-3 TCP

Until ns-3.10 release, ns-3 contained a port of the TCP model from GTNetS. This implementation was substantially rewritten by Adriam Tam for ns-3.10. The model is a full TCP, in that it is bidirectional and attempts to model the connection setup and close logic.

The implementation of TCP is contained in the following files:

```plaintext
src/internet/model/tcp-header.{cc,h}
src/internet/model/tcp-14-protocol.{cc,h}
src/internet/model/tcp-socket-factory-impl.{cc,h}
src/internet/model/tcp-socket-base.{cc,h}
src/internet/model/tcp-socket-factory.{cc,h}
src/internet/model/tcp-tx-buffer.{cc,h}
src/internet/model/tcp-rx-buffer.{cc,h}
src/internet/model/tcp-tahoe.{cc,h}
src/internet/model/tcp-reno.{cc,h}
src/internet/model/tcp-newreno.{cc,h}
src/internet/model/rtt-estimator.{cc,h}
src/network/model/sequence-number.{cc,h}
```

Different variants of TCP congestion control are supported by subclassing the common base class TcpSocketBase. Several variants are supported, including RFC 793 (no congestion control), Tahoe, Reno, and NewReno. NewReno is used by default.

#### Usage

In many cases, usage of TCP is set at the application layer by telling the ns-3 application which kind of socket factory to use.

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

```c
// 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.

To configure behavior of TCP, a number of parameters are exported through the Attributes. These are documented in the Doxygen <http://www.nsnam.org/doxygen/classns3_1_1_tcp_socket.html> for class TcpSocket. For example, the maximum segment size is a settable attribute.

To set the default socket type before any internet stack-related objects are created, one may put the following statement at the top of the simulation program:

Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue ("ns3::TcpTahoe"));

For users who wish to have a pointer to the actual socket (so that socket operations like Bind(), setting socket options, etc. can be done on a per-socket basis), Tcp sockets can be created by using the Socket::CreateSocket() method. The TypeId passed to CreateSocket() must be of type ns3::SocketFactory, so configuring the underlying socket type must be done by twiddling the attribute associated with the underlying TcpL4Protocol object. The easiest way to get at this would be through the attribute configuration system. In the below example, the Node container “n0n1” is accessed to get the zeroth element, and a socket is created on this node:

// Create and bind the socket...
TypeId tid = TypeId::LookupByName ("ns3::TcpTahoe");
Config::Set ("/NodeList/*/$/ns3::TcpL4Protocol/SocketType", TypeIdValue (tid));
Ptr<Socket> localSocket =
    Socket::CreateSocket (n0n1.Get (0), TcpSocketFactory::GetTypeId ());

Above, the “*” wild card for node number is passed to the attribute configuration system, so that all future sockets on all nodes are set to Tahoe, not just on node ‘n0n1.Get (0)’. If one wants to limit it to just the specified node, one would have to do something like:

// Create and bind the socket...
TypeId tid = TypeId::LookupByName ("ns3::TcpTahoe");
std::stringstream nodeId; 
nodeId << n0n1.Get (0)->GetId ();
std::string specificNode = "/*/\$ns3::TcpL4Protocol/SocketType";
Config::Set (specificNode, TypeIdValue (tid));
Ptr<Socket> localSocket =
    Socket::CreateSocket (n0n1.Get (0), TcpSocketFactory::GetTypeId ());

Once a TCP socket is created, one 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). Sockets API for a review of how sockets are used in ns-3.

Validation

Several TCP validation test results can be found in the wiki page describing this implementation.

Current limitations

- Only IPv4 is supported
- Neither the Nagle algorithm nor SACK are supported

12.5. TCP models in ns-3
12.5.3 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.

Prerequisites

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

Building NSC requires the packages flex and bison.

Configuring and Downloading

Using the build.py script in ns-3-allinone directory, NSC will be enabled by default unless the platform does not support it. To disable it when building ns-3, type::

```bash
./waf configure --enable-examples --enable-tests --disable-nsc
```

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 following ns-3 test suite::

```bash
./test.py -s ns3-tcp-interoperability
```

If NSC has been successfully built, the following test should show up in the results::

```
PASS TestSuite ns3-tcp-interoperability
```

This confirms that NSC is ready to use.

Usage

There are a few example files. Try:

```bash
./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/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.SetNs3Stack("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 SetNs3Stack. 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 calling 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:

```bash
~/ns-3.10> 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.

**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 Attributes system, via a sysctl-like interface. In the examples/tcp/tcp-nsc-zoo example, you can see the following configuration:

```cpp
// this disables TCP SACK, wscale and timestamps on node 1 (the attributes represent sysct1-values).
Config::Set("/NodeList/1/$ns3::Ns3Ns3Stack<linux2.6.26>/net.ipv4.tcp_sack",
           StringValue("0"));
Config::Set("/NodeList/1/$ns3::Ns3Ns3Stack<linux2.6.26>/net.ipv4.tcp_timestamps",
           StringValue("0"));
Config::Set("/NodeList/1/$ns3::Ns3Ns3Stack<linux2.6.26>/net.ipv4.tcp_window_scaling",
           StringValue("0"));
```

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

**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.

### 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/model/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/model/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-14, 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-14 class, when the nsc-tcp-14 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.

### Limitations

- NSC only works on single-interface nodes; attempting to run it on a multi-interface node will cause a program error.
- Cygwin and OS X PPC are not supported
- The non-Linux stacks of NSC are not supported in ns-3
- Not all socket API callbacks are supported

For more information, see this wiki page.
This chapter describes the ns-3 ns3::LteNetDevice and related models. By adding ns3::LteNetDevice objects to ns-3 nodes, one can create models of 3GPP E-UTRAN infrastructure and Long Term Evolution (LTE) networks.

Below, we list some more details about what ns-3 LTE models cover but, in summary, the most important features of the ns-3 model are:

- a basic implementation of both the User Equipment (UE) and the enhanced NodeB (eNB) devices,
- RRC entities for both the UE and the eNB,
- a state-of-the-art Adaptive Modulation and Coding (AMC) scheme for the downlink
- the management of the data radio bearers (with their QoS parameters), the MAC queues and the RLC instances,
- Channel Quality Indicator (CQI) management,
- support for both uplink and downlink packet scheduling,
- a PHY layer model with Resource Block level granularity
- a channel model with the outdoor E-UTRAN propagation loss model.

13.1 Model Description

The source code for the LTE models lives in the directory src/lte.

13.1.1 Design

The LTE model provides a basic implementation of LTE devices, including propagation models and PHY and MAC layers. It allow to simulate an E-UTRAN interface where one eNB and several UEs can communicate among them using a shared downlink (uplink) channel.

The PHY layer has been developed extending the Spectrum Framework \(^1\). The MAC model, instead, has been developed extending and adding some features to the base class ns3::NetDevice.

\(^1\) N. Baldo and M. Miozzo, Spectrum-aware Channel and PHY layer modeling for ns3, Proceedings of ICST NSTools 2009, Pisa, Italy. The framework is designed to simulate only data transmissions. For the transmission of control messages (such as CQI feedback, PDCCH, etc..) will be used an ideal control channel.
Physical layer

A ns3::LtePhy class models the LTE PHY layer.

Basic functionalities of the PHY layer are: (i) transmit packets coming from the device to the channel; (ii) receive packets from the channel; (ii) evaluate the quality of the channel from the Signal To Noise ratio of the received signal; and (iii) forward received packets to the device.

Both the PHY and channel have been developed extending ns3::SpectrumPhy and ns3::SpectrumChannel classes, respectively.

The module implements an FDD channel access. In FDD channel access, downlink and uplink transmissions work together in the time but using a different set of frequencies. Since DL and UL are independent between them, the PHY is composed by couple of ns3::LteSpectrumPhy object (i.e., implemented into the ns3::LteSpectrumPhy class); one for the downlink and one for the uplink. The ns3::LtePhy stores and manages both downlink and uplink ns3::LteSpectrumPhy elements.

In order to customize all physical functionalities for both UE and eNB devices, dedicated classes have been inherited from ones described before. In particular, ns3::UePhy and ns3::EnbPhy classes, inherited from the ns3::LtePhy class, implement the PHY layer for the UE and the eNB, respectively. In the same way, ns3::UeLteSpectrumPhy and ns3::EnbLteSpectrumPhy classes, inherited from the ns3::LteSpectrumPhy, implement the downlink/uplink spectrum channel for the UE and the eNB, respectively.

The figure below shows how UE and eNB can exchange packets through the considered PHY layer.

![Figure 13.1: DL and UL transmission on the LTE network](image)

For the downlink, when the eNB wants to send packets, it calls the StartTx function to send them into the downlink channel. Then, the downlink channel delivers the burst of packets to all the ns3::UeLteSpectrumPhy attached to it, handling the StartRx function. When the UE receives packets, it executes the following tasks:
• compute the SINR for all the sub channel used in the downlink
• create and send CQI feedbacks
• forward all the received packets to the device

The uplink works similarly to the previous one.

Propagation Loss Models

A proper propagation loss model has been developed for the LTE E-UTRAN interface (see \(^2\) and \(^3\)). It is used by the PHY layer to compute the loss due to the propagation.

The LTE propagation loss model is composed by 4 different models (shadowing, multipath, penetration loss and path loss) \(^2\):

- Pathloss: \( PL = 128.1 + (37.6 \times \log_{10}(R)) \), where \( R \) is the distance between the UE and the eNB in Km.
- Multipath: Jakes model
- PenetrationLoss: 10 dB
- Shadowing: log-normal distribution (mean=0dB, standard deviation=8dB)

Every time that the LteSpectrumPHY::StartRx () function is called, the SpectrumInterferenceModel is used to computed the SINR, as proposed in \(^3\). Then, the network device uses the AMC module to map the SINR to a proper CQI and to send it to the eNB using the ideal control channel.

LTE Devices

All the common functionalities of the LTE network devices have been defined into the ns3::LteNetDevice class. Moreover, the LTE device has been conceived as a container of several entities such as MAC, RRC, RLC etc .. For each of these entity a dedicated class has been developed.

For each device are defined the following entity/element

- the LTE PHY layer (described in the previous sub section)
- rrc entity
- mac entity
- rlc entity

The module is perfectly integrated into the whole ns-3 project: it is already possible to attach to each device a TCP/IP protocol stack and all the implemented applications (i.e., udp client/server, trace based, etc..).

The RRC Entity

RRC entity is implemented by the ns3::RrcEntity class, and provides only the Radio Bearer management functionality. A dedicated bearer is created for each downlink flow.

The RRC entity performs the classification of the packets coming from the upper layer into the corresponding Radio Bearer. This classification is based on the information provided by the class ns3::IpcsClassifierRecord.

\(^2\) 3GPP TS 25.814 (http://www.3gpp.org/ftp/specs/html-info/25814.htm)
The MAC Entity

Class `ns3::MacEntity` provides a basic implementation of the MAC entity for the LTE device. Moreover, `ns3::EnbMacEntity` and `ns3::UeMacEntity` classes, inherited from the previous one, provides an implementation for the eNB and the UE MAC entity, respectively. In all MAC entities is defined the AMC module \(^4\). Furthermore, into the `ns3::EnbMacEntity` class are defined also both uplink and downlink schedulers.

Every time the PHY layer of the UE receives a packet form the channel, it calls the AMC module, define into the MAC entity, in order to convert the SINR of the received signal to CQI feedbacks. Every sub frame, the eNB performs both uplink and downlink radio resource allocation. Actually only a simple packet scheduler has been implemented that is able to send, every sub frame, only one packet in the downlink.

The RLC Entity

The RLC entity performs an interface between the MAC layer and the MAC queue for a given bearer. Actually, only the RLC Transport Mode has been implemented.

Control Channel

Control channel keeps a very important role in LTE networks. In fact, it is responsible of the transmission of several information (i.e., CQI feedback, allocation map, etc...). For this reason, also in a framework oriented to data transmission, it is necessary to find a technique for exchange these information. To this aim, an ideal control channel will be developed. Using ideal control messages, both UE and eNB can exchange control information without simulating a realistic transmission over the LTE channel.

Two types of control messages have been implemented: the Pdcch Map Ideal Control Message and the Cqi Ideal Control Message. The first one is used by the eNB to send the uplink and downlink resource mapping to all registered UE. In particular, this message carries information about UEs that have been scheduled in the ul/dl, a list of assigned sub channels and the selected MCS for the transmission. The second one, instead, is used by the UE to send CQI feedbacks to the eNB.

13.1.2 Scope and Limitations

The framework has been designed in order to support data transmission for both uplink and downlink. It is important to note that downlink and uplink transmissions are managed by the packet scheduler that works at the eNB. It decides, in fact, what UEs should be scheduled every TTI and what radio resource should be allocated to them.

In the current implementation, the downlink transmission is administrated by the downlink packet scheduler. Furthermore, no packet scheduler for uplink transmission has been developed. As a consequence, for the downlink packet are sent only after scheduling decisions; for the uplink, instead, packet are sent directly, without any scheduling decisions. Finally, the transmission of control messages (such as CQI feedbacks, PDCCH, etc..) is done by an ideal control channel.

13.1.3 Future Work

In the future, several LTE features will be developed in order to improve the proposed module.

In particular, for the near future have been scheduled the following implementations:

- a more efficient design for the RRM (Radio resource management)

\(^4\) 3GPP R1-081483 (available on http://www.3gpp.org/ftp/tsg_ran/WG1_RL1/TSGR1_52b/Docs/R1-081483.zip )
• a complete packet scheduler (i.e., a simple round robin scheme, maximum throughput and proportional fair allocation schemes) for both downlink and uplink, in order to support a standard compliant packet transmission
• ideal PDCCH control messages
• a standard compliant RLC entity
• PHY error model

13.1.4 References

13.2 Usage

The main way that users who write simulation scripts will typically interact with the LTE models is through the helper API and through the publicly visible attributes of the model.

The helper API is defined in src/lte/helper/lte-helper.h.

The example src/lte/examples/ contain some basic code that shows how to set up the model in order to simulate an E-UTRAN downlink transmission.

13.2.1 Examples

src/lte/examples/lte-device.cc shows how it is possible to set up the LTE module:

```cpp
NodeContainer ueNodes;
NodeContainer enbNodes;

ueNodes.Create (1);
enbNodes.Create (1);

LteHelper lte;

NetDeviceContainer ueDevs, enbDevs;
ueDevs = lte.Install (ueNodes, LteHelper::DEVICE_TYPE_USER_EQUIPMENT);
enbDevs = lte.Install (enbNodes, LteHelper::DEVICE_TYPE_ENODEB);
```

The helper method ns3::LteHelper::Install() creates LTE device, the DL, UL physical layer and attach the to proper LTE channels.

Moreover, to simulate a complete LTE system, it is necessary to define other information, as expressed in what follows:

1. install IP protocol stack:

   ```cpp
   InternetStackHelper stack;
   stack.Install (ueNodes);
   stack.Install (enbNodes);
   Ipv4AddressHelper address;
   address.SetBase ("10.1.1.0", "255.255.255.0");
   Ipv4InterfaceContainer UEinterfaces = address.Assign (ueDevs);
   Ipv4InterfaceContainer ENBinterface = address.Assign (enbDevs);
   ```

2. register UE to a given eNB:

   ```cpp
   Ptr<EnbNetDevice> enb = enbDevs.Get (0)->GetObject<EnbNetDevice> ()
   Ptr<UeNetDevice> ue = ueDevs.Get (i)->GetObject<UeNetDevice> ()
   lte.RegisterUeToTheEnb (ue, enb);
   ```
3. create the mobility model for each device:

```cpp
Ptr<ConstantPositionMobilityModel> enbMobility = new ConstantPositionMobilityModel();
enbMobility->SetPosition (Vector (0.0, 0.0, 0.0));
lte.AddMobility (enb->GetPhy (), enbMobility);

Ptr<ConstantVelocityMobilityModel> ueMobility = new ConstantVelocityMobilityModel();
ueMobility->SetPosition (Vector (30.0, 0.0, 0.0));
ueMobility->SetVelocity (Vector (30.0, 0.0, 0.0));
lte.AddMobility (ue->GetPhy (), ueMobility);
```

4. define a set of sub channels to use for dl and ul transmission:

```cpp
std::vector<int> dlSubChannels;
for (int i = 0; i < 25; i++)
{
    dlSubChannels.push_back (i);
}

std::vector<int> ulSubChannels;
for (int i = 50; i < 100; i++)
{
    ulSubChannels.push_back (i);
}

enb->GetPhy ()->SetDownlinkSubChannels (dlSubChannels);
enb->GetPhy ()->SetUplinkSubChannels (ulSubChannels);
ue->GetPhy ()->SetDownlinkSubChannels (dlSubChannels);
ue->GetPhy ()->SetUplinkSubChannels (ulSubChannels);
```

5. define a channel realization for the PHY model:

```cpp
lte.AddDownlinkChannelRealization (enbMobility, ueMobility, ue->GetPhy ());
```

### 13.2.2 Helpers

### 13.2.3 Attributes

### 13.2.4 Tracing

### 13.2.5 Logging

### 13.2.6 Caveats

### 13.3 Validation

In the `src/lte/example/lte-amc.cc` has been developed an important example that shows the proper functioning of both AMC module and Channel model. The analyzed scenario is composed by two nodes: a eNB and a single UE (registered to the eNB). The UE moves into the cell using the `ns3::ConstantVelocityMobilityModel`, along a radial direction. The proposed example describes how the channel quality decreases as the distance between UE and eNB increases. As a consequence, the total bit rate (in bits per TTI) available to the UE decreases as the distance between nodes increases, as expected.
CHAPTER
FOURTEEN

MESH NETDEVICE

Placeholder chapter

The Mesh NetDevice based on 802.11s was added in ns-3.6. An overview presentation by Kirill Andreev was published at the wns-3 workshop in 2009: http://www.nsnam.org/wiki/index.php/Wns3-2009.
Parallel and distributed discrete event simulation allows the execution of a single simulation program on multiple processors. By splitting up the simulation into logical processes, LPs, each LP can be executed by a different processor. This simulation methodology enables very large-scale simulations by leveraging increased processing power and memory availability. In order to ensure proper execution of a distributed simulation, message passing between LPs is required. To support distributed simulation in ns-3, the standard Message Passing Interface (MPI) is used, along with a new distributed simulator class. Currently, dividing a simulation for distributed purposes in ns-3 can only occur across point-to-point links.

15.1 Current Implementation Details

During the course of a distributed simulation, many packets must cross simulator boundaries. In other words, a packet that originated on one LP is destined for a different LP, and in order to make this transition, a message containing the packet contents must be sent to the remote LP. Upon receiving this message, the remote LP can rebuild the packet and proceed as normal. The process of sending an receiving messages between LPs is handled easily by the new MPI interface in ns-3.

Along with simple message passing between LPs, a distributed simulator is used on each LP to determine which events to process. It is important to process events in time-stamped order to ensure proper simulation execution. If a LP receives a message containing an event from the past, clearly this is an issue, since this event could change other events which have already been executed. To address this problem, a conservative synchronization algorithm with lookahead is used in ns-3. For more information on different synchronization approaches and parallel and distributed simulation in general, please refer to “Parallel and Distributed Simulation Systems” by Richard Fujimoto.

15.1.1 Remote point-to-point links

As described in the introduction, dividing a simulation for distributed purposes in ns-3 currently can only occur across point-to-point links; therefore, the idea of remote point-to-point links is very important for distributed simulation in ns-3. When a point-to-point link is installed, connecting two nodes, the point-to-point helper checks the system id, or rank, of both nodes. The rank should be assigned during node creation for distributed simulation and is intended to signify on which LP a node belongs. If the two nodes are on the same rank, a regular point-to-point link is created. If, however, the two nodes are on different ranks, then these nodes are intended for different LPs, and a remote point-to-point link is used. If a packet is to be sent across a remote point-to-point link, MPI is used to send the message to the remote LP.
15.1.2 Distributing the topology

Currently, the full topology is created on each rank, regardless of the individual node system ids. Only the applications are specific to a rank. For example, consider node 1 on LP 1 and node 2 on LP 2, with a traffic generator on node 1. Both node 1 and node 2 will be created on both LP1 and LP2; however, the traffic generator will only be installed on LP1. While this is not optimal for memory efficiency, it does simplify routing, since all current routing implementations in ns-3 will work with distributed simulation.

15.2 Running Distributed Simulations

15.2.1 Prerequisites

Ensure that MPI is installed, as well as mpic++. In Ubuntu repositories, these are openmpi-bin, openmpi-common, openmpi-doc, libopenmpi-dev. In Fedora, these are openmpi and openmpi-devel.

Note:
There is a conflict on some Fedora systems between libotf and openmpi. A possible “quick-fix” is to yum remove libotf before installing openmpi. This will remove conflict, but it will also remove emacs. Alternatively, these steps could be followed to resolve the conflict::

1) Rename the tiny otfdump which emacs says it needs:

   mv /usr/bin/otfdump /usr/bin/otfdump.emacs-version

2) Manually resolve openmpi dependencies:

   sudo yum install libgfortran libtorque numactl

3) Download rpm packages:

   openmpi-1.3.1-1.fc11.i586.rpm
   openmpi-devel-1.3.1-1.fc11.i586.rpm
   openmpi-libs-1.3.1-1.fc11.i586.rpm
   openmpi-vt-1.3.1-1.fc11.i586.rpm

   from

   http://mirrors.kernel.org/fedora/releases/11/Everything/i386/os/Packages/

4) Force the packages in:

   sudo rpm -ivh --force openmpi-1.3.1-1.fc11.i586.rpm
   openmpi-libs-1.3.1-1.fc11.i586.rpm openmpi-devel-1.3.1-1.fc11.i586.rpm
   openmpi-vt-1.3.1-1.fc11.i586.rpm

Also, it may be necessary to add the openmpi bin directory to PATH in order to execute mpic++ and mpirun from the command line. Alternatively, the full path to these executables can be used. Finally, if openmpi complains about the inability to open shared libraries, such as libmpi_cxx.so.0, it may be necessary to add the openmpi lib directory to LD_LIBRARY_PATH.

15.2.2 Building and Running Examples

If you already built ns-3 without MPI enabled, you must re-build::
./waf distclean

Configure ns-3 with the --enable-mpi option:

```
./waf -d debug configure --enable-examples --enable-tests --enable-mpi
```

Ensure that MPI is enabled by checking the optional features shown from the output of configure.

Next, build ns-3:

```
./waf
```

After building ns-3 with mpi enabled, the example programs are now ready to run with mpirun. Here are a few examples (from the root ns-3 directory):

```
mpirun -np 2 ./waf --run simple-distributed
mpirun -np 4 -machinefile mpihosts ./waf --run 'nms-udp-nix --LAN=2 --CN=4 --nix=1'
```

The np switch is the number of logical processors to use. The machinefile switch is which machines to use. In order to use machinefile, the target file must exist (in this case mpihosts). This can simply contain something like:

```
localhost
localhost
localhost
```

... Or if you have a cluster of machines, you can name them.

NOTE: Some users have experienced issues using mpirun and waf together. An alternative way to run distributed examples is shown below:

```
./waf shell
cd build/debug
mpirun -np 2 src/mpi/examples/simple-distributed
```

### 15.2.3 Creating custom topologies

The example programs in src/mpi/examples give a good idea of how to create different topologies for distributed simulation. The main points are assigning system ids to individual nodes, creating point-to-point links where the simulation should be divided, and installing applications only on the LP associated with the target node.

Assigning system ids to nodes is simple and can be handled two different ways. First, a NodeContainer can be used to create the nodes and assign system ids:

```
NodeContainer nodes;
nodes.Create (5, 1); // Creates 5 nodes with system id 1.
```

Alternatively, nodes can be created individually, assigned system ids, and added to a NodeContainer. This is useful if a NodeContainer holds nodes with different system ids:

```
NodeContainer nodes;
Ptr<Node> node1 = CreateObject<Node> (0); // Create node1 with system id 0
Ptr<Node> node2 = CreateObject<Node> (1); // Create node2 with system id 1
nodes.Add (node1);
nodes.Add (node2);
```

Next, where the simulation is divided is determined by the placement of point-to-point links. If a point-to-point link is created between two nodes with different system ids, a remote point-to-point link is created, as described in *Current Implementation Details*. 

---

**15.2. Running Distributed Simulations**

73
Finally, installing applications only on the LP associated with the target node is very important. For example, if a traffic generator is to be placed on node 0, which is on LP0, only LP0 should install this application. This is easily accomplished by first checking the simulator system id, and ensuring that it matches the system id of the target node before installing the application.

### 15.3 Tracing During Distributed Simulations

Depending on the system id (rank) of the simulator, the information traced will be different, since traffic originating on one simulator is not seen by another simulator until it reaches nodes specific to that simulator. The easiest way to keep track of different traces is to just name the trace files or pcaps differently, based on the system id of the simulator. For example, something like this should work well, assuming all of these local variables were previously defined::

```cpp
if (MpiInterface::GetSystemId () == 0)
{
    pointToPoint.EnablePcapAll ("distributed-rank0");
    phy.EnablePcap ("distributed-rank0", apDevices.Get (0));
    csma.EnablePcap ("distributed-rank0", csmaDevices.Get (0), true);
}
else if (MpiInterface::GetSystemId () == 1)
{
    pointToPoint.EnablePcapAll ("distributed-rank1");
    phy.EnablePcap ("distributed-rank1", apDevices.Get (0));
    csma.EnablePcap ("distributed-rank1", csmaDevices.Get (0), true);
}
```
CHAPTER SIXTEEN

NETWORK MODULE

16.1 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

Each network packet contains a byte buffer, a set of byte tags, a set of packet tags, and metadata. The byte buffer stores the serialized content of the headers and trailers added to a packet. The serialized representation of these headers 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.

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.

One problem that this design choice raises is that it is difficult to pretty-print the packet headers without context. The packet metadata describes the type of the headers and trailers which were serialized in the byte buffer. The maintenance of metadata is optional and disabled by default. To enable it, you must call Packet::EnableMetadata() and this will allow you to get non-empty output from Packet::Print and Packet::Print.

Also, developers often want to store data in packet objects that is not found in the real packets (such as timestamps or flow-ids). The Packet class deals with this requirement by storing a set of tags (class Tag). We have found two classes of use cases for these tags, which leads to two different types of tags. So-called ‘byte’ tags are used to tag a subset of the bytes in the packet byte buffer while ‘packet’ tags are used to tag the packet itself. The main difference between these two kinds of tags is what happens when packets are copied, fragmented, and reassembled: ‘byte’ tags follow bytes while ‘packet’ tags follow packets. Another important difference between these two kinds of tags is that byte tags cannot be removed and are expected to be written once, and read many times, while packet tags are expected to be written once, read many times, and removed exactly once. An example of a ‘byte’ tag is a FlowIdTag which contains a flow id and is set by the application generating traffic. An example of a ‘packet’ tag is a cross-layer QoS class id set by an application and processed by a lower-level MAC layer.

Memory management of Packet objects is entirely automatic and extremely efficient: memory for the application-level payload can be modeled by a virtual buffer of zero-filled bytes for which memory is never allocated unless explicitly
requested by the user or unless the packet is fragmented or serialized out to a real network device. 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.

### 16.1.1 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, a list of byte Tags, a list of packet Tags, and a PacketMetadata object:

- 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.

- Each 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).

![Figure 16.1: Implementation overview of Packet class.](image)

Figure Implementation overview of Packet class. is a high-level overview of the Packet implementation; more detail on the byte Buffer implementation is provided later in Figure Implementation overview of a packet’s byte Buffer. In ns-3, the Packet byte buffer is analogous to a Linux skbuff or BSD mbuf; it is a serialized representation of the actual data in the packet. The tag lists are containers 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.

Packets are reference counted objects. They are handled with smart pointer (Ptr) objects like many of the objects in the ns-3 system. One small difference you will see is that class Packet does not inherit from class Object or class RefCountBase, and implements the Ref() and Unref() methods directly. This was designed to avoid the overhead of a vtable in class Packet.

The Packet class is designed to be copied cheaply; the overall design is based on Copy on Write (COW). When there are multiple references to a packet object, and there is an operation on one of them, only so-called “dirty” operations will trigger a deep copy of the packet:

- ns3::Packet::AddHeader()
- ns3::Packet::AddTrailer()
- both versions of ns3::Packet::AddAtEnd()
- Packet::RemovePacketTag()

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 lists. 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.

16.1.2 Using the packet interface

This section describes how to create and use the ns3::Packet object.

Creating a new packet

The following command will create a new packet with a new unique Id.:

```c++
Ptr<Packet> pkt = Create<Packet> ();
```

What is the Uid (unique Id)? It is an internal id that the system uses to identify packets. It can be fetched via the following method:

```c++
uint32_t uid = pkt->GetUid ();
```
But please note the following. This uid is an internal uid and cannot be counted on to provide an accurate counter of how many “simulated packets” of a particular protocol are in the system. It is not trivial to make this uid into such a counter, because of questions such as what should the uid be when the packet is sent over broadcast media, or when fragmentation occurs. If a user wants to trace actual packet counts, he or she should look at e.g. the IP ID field or transport sequence numbers, or other packet or frame counters at other protocol layers.

We mentioned above that it is possible to create packets with zero-filled payloads that do not actually require a memory allocation (i.e., the packet may behave, when delays such as serialization or transmission delays are computed, to have a certain number of payload bytes, but the bytes will only be allocated on-demand when needed). The command to do this is, when the packet is created::

```c
Ptr<Packet> pkt = Create<Packet> (N);
```

where N is a positive integer.

The packet now has a size of N bytes, which can be verified by the GetSize() method::

```c
/**
 * \returns the size in bytes of the packet (including the zero-filled initial payload)
 */
uint32_t GetSize (void) const;
```

You can also initialize a packet with a character buffer. The input data is copied and the input buffer is untouched. The constructor applied is::

```c
Packet (uint8_t const *buffer, uint32_t size);
```

Here is an example::

```c
Ptr<Packet> pkt1 = Create<Packet> (reinterpret_cast<const uint8_t*> ("hello"), 5);
```

Packets are freed when there are no more references to them, as with all ns-3 objects referenced by the Ptr class.

### Adding and removing Buffer data

After the initial packet creation (which may possibly create some fake initial bytes of payload), all subsequent buffer data is added by adding objects of class Header or class Trailer. Note that, even if you are in the application layer, handling packets, and want to write application data, you write it as an ns3::Header or ns3::Trailer. If you add a Header, it is prepended to the packet, and if you add a Trailer, it is added to the end of the packet. If you have no data in the packet, then it makes no difference whether you add a Header or Trailer. Since the APIs and classes for header and trailer are pretty much identical, we’ll just look at class Header here.

The first step is to create a new header class. All new Header classes must inherit from class Header, and implement the following methods:

- `Serialize ()`
- `Deserialize ()`
- `GetSerializedSize ()`
- `Print ()`

To see a simple example of how these are done, look at the UdpHeader class headers `src/internet/model/udp-header.cc`. There are many other examples within the source code.

Once you have a header (or you have a preexisting header), the following Packet API can be used to add or remove such headers:
/**
 * Add header to this packet. This method invokes the
 * Header::GetSerializedSize and Header::Serialize
 * methods to reserve space in the buffer and request the
 * header to serialize itself in the packet buffer.
 * *
 * \param header a reference to the header to add to this packet.
 */
void AddHeader (const Header & header);
/**
 * Deserialize and remove the header from the internal buffer.
 * This method invokes Header::Deserialize.
 * *
 * \param header a reference to the header to remove from the internal buffer.
 * \returns the number of bytes removed from the packet.
 */
uint32_t RemoveHeader (Header &header);
/**
 * Deserialize but does _not_ remove the header from the internal buffer.
 * This method invokes Header::Deserialize.
 * *
 * \param header a reference to the header to read from the internal buffer.
 * \returns the number of bytes read from the packet.
 */
uint32_t PeekHeader (Header &header) const;

For instance, here are the typical operations to add and remove a UDP header:

```c++
// add header
Ptr<Packet> packet = Create<Packet> () ;
UdpHeader udpHeader ;
// Fill out udpHeader fields appropriately
packet->AddHeader (udpHeader) ;
...
// remove header
UdpHeader udpHeader ;
packet->RemoveHeader (udpHeader) ;
// Read udpHeader fields as needed
```

**Adding and removing Tags**

There is a single base class of Tag that all packet tags must derive from. They are used in two different tag lists in the
packet; the lists have different semantics and different expected use cases.

As the names imply, ByteTags follow bytes and PacketTags follow packets. What this means is that when operations
are done on packets, such as fragmentation, concatenation, and appending or removing headers, the byte tags keep
track of which packet bytes they cover. For instance, if a user creates a TCP segment, and applies a ByteTag to the
segment, each byte of the TCP segment will be tagged. However, if the next layer down inserts an IPv4 header, this
ByteTag will not cover those bytes. The converse is true for the PacketTag; it covers a packet despite the operations
on it.

PacketTags are limited in size to 20 bytes. This is a modifiable compile-time constant in
src/network/model/packet-tag-list.h. ByteTags have no such restriction.

Each tag type must subclass ns3::Tag, and only one instance of each Tag type may be in each tag list. Here are a
few differences in the behavior of packet tags and byte tags.
• **Fragmentation:** As mentioned above, when a packet is fragmented, each packet fragment (which is a new packet) will get a copy of all packet tags, and byte tags will follow the new packet boundaries (i.e. if the fragmented packets fragment across a buffer region covered by the byte tag, both packet fragments will still have the appropriate buffer regions byte tagged).

• **Concatenation:** When packets are combined, two different buffer regions will become one. For byte tags, the byte tags simply follow the respective buffer regions. For packet tags, only the tags on the first packet survive the merge.

• **Finding and Printing:** Both classes allow you to iterate over all of the tags and print them.

• **Removal:** Users can add and remove the same packet tag multiple times on a single packet (AddPacketTag () and RemovePacketTag ()). The packet However, once a byte tag is added, it can only be removed by stripping all byte tags from the packet. Removing one of possibly multiple byte tags is not supported by the current API.

As of ns-3.5 and later, Tags are not serialized and deserialized to a buffer when Packet::Serialize () and Packet::Deserialize () are called; this is an open bug.

If a user wants to take an existing packet object and reuse it as a new packet, he or she should remove all byte tags and packet tags before doing so. An example is the UdpEchoServer class, which takes the received packet and “turns it around” to send back to the echo client.

The Packet API for byte tags is given below:

```cpp
/** 
 * \param tag the new tag to add to this packet
 * 
 * Tag each byte included in this packet with the
 * new tag.
 * 
 * Note that adding a tag is a const operation which is pretty
 * un-intuitive. The rationale is that the content and behavior of
 * a packet is _not_ changed when a tag is added to a packet: any
 * code which was not aware of the new tag is going to work just
 * the same if the new tag is added. The real reason why adding a
 * tag was made a const operation is to allow a trace sink which gets
 * a packet to tag the packet, even if the packet is const (and most
 * trace sources should use const packets because it would be
 * totally evil to allow a trace sink to modify the content of a
 * packet).
 */
void AddByteTag (const Tag &tag) const;
/**
 * \returns an iterator over the set of byte tags included in this packet.
 */
ByteTagIterator GetByteTagIterator (void) const;
/**
 * \param tag the tag to search in this packet
 * \returns true if the requested tag type was found, false otherwise.
 * 
 * If the requested tag type is found, it is copied in the user’s
 * provided tag instance.
 */
bool FindFirstMatchingByteTag (Tag &tag) const;
/**
 * Remove all the tags stored in this packet.
 */
void RemoveAllByteTags (void);
```
void PrintByteTags (std::ostream &os) const;

The Packet API for packet tags is given below:

void AddPacketTag (const Tag &tag) const;

bool RemovePacketTag (Tag &tag);

bool PeekPacketTag (Tag &tag) const;

void RemoveAllPacketTags (void);

void PrintPacketTags (std::ostream &os) const;

PacketTagIterator GetPacketTagIterator (void) const;
Here is a simple example illustrating the use of tags from the code in `src/internet/model/udp-socket-impl.cc`:

```cpp
Ptr<Packet> p; // pointer to a pre-existing packet
SocketIpTtlTag tag
tag.SetTtl (m_ipMulticastTtl); // Convey the TTL from UDP layer to IP layer
p->AddPacketTag (tag);
```

This tag is read at the IP layer, then stripped (`src/internet/model/ipv4-l3-protocol.cc`):

```cpp
uint8_t ttl = m_defaultTtl;
SocketIpTtlTag tag;
bool found = packet->RemovePacketTag (tag);
if (found)
  {  
    ttl = tag.GetTtl ();
  }
```

### Fragmentation and concatenation

Packets may be fragmented or merged together. For example, to fragment a packet `p` of 90 bytes into two packets, one containing the first 10 bytes and the other containing the remaining 80, one may call the following code:

```cpp
Ptr<Packet> frag0 = p->CreateFragment (0, 10);
Ptr<Packet> frag1 = p->CreateFragment (10, 90);
```

As discussed above, the packet tags from `p` will follow to both packet fragments, and the byte tags will follow the byte ranges as needed.

Now, to put them back together:

```cpp
frag0->AddAtEnd (frag1);
```

Now `frag0` should be equivalent to the original packet `p`. If, however, there were operations on the fragments before being reassembled (such as tag operations or header operations), the new packet will not be the same.

### Enabling metadata

We mentioned above that packets, being on-the-wire representations of byte buffers, present a problem to print out in a structured way unless the printing function has access to the context of the header. For instance, consider a tcpdump-like printer that wants to pretty-print the contents of a packet.

To enable this usage, packets may have metadata enabled (disabled by default for performance reasons). This class is used by the Packet class to record every operation performed on the packet’s buffer, and provides an implementation of `Packet::Print()` method that uses the metadata to analyze the content of the packet’s buffer.

The metadata is also used to perform extensive sanity checks at runtime when performing operations on a Packet. For example, this metadata is used to verify that when you remove a header from a packet, this same header was actually present at the front of the packet. These errors will be detected and will abort the program.

To enable this operation, users will typically insert one or both of these statements at the beginning of their programs:

```cpp
Packet::EnablePrinting ();
Packet::EnableChecking ();
```
16.1.3 Sample programs


16.1.4 Implementation details

Private member variables

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

```c
Buffer m_buffer;
ByteTagList m_byteTagList;
PacketTagList m_packetTagList;
PacketMetadata m_metadata;
mutable uint32_t m_refCount;
static uint32_t m_globalUid;
```

Each Packet has a Buffer and two Tags lists, a PacketMetadata object, and a ref count. A static member variable keeps track of the UIDs allocated. The actual uid of the packet is stored in the PacketMetadata.

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.

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:

```c
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];
};
```

- `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 16.2: Implementation overview of a packet’s byte Buffer.

This data structure is summarized in Figure Implementation overview of a packet’s byte Buffer. 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.

**Tags implementation**

(XXX revise me)

Tags are implemented by a single pointer which points to the start of a linked list of TagData 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.)
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 between 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:

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

**Memory management**

Describe dataless vs. data-full packets.

**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:

- ns3::Packet::AddHeader
- ns3::Packet::AddTrailer
- both versions of ns3::Packet::AddAtEnd
- ns3::Packet::RemovePacketTag

Non-dirty operations:

- ns3::Packet::AddPacketTag
- ns3::Packet::PeekPacketTag
- ns3::Packet::RemoveAllPacketTags
- ns3::Packet::AddByteTag
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.

16.2 Node and NetDevices Overview

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

![High-level node architecture](image-url)

**Figure 16.3: High-level node architecture**
In ns-3, nodes are instances of ns3::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. High-level node architecture illustrates that ns3::Node objects contain a list of ns3::Application instances (initially, the list is empty), a list of ns3::NetDevice instances (initially, the list is empty), a list of ns3::Node::ProtocolHandler instances, a unique integer ID, and a system ID (for distributed simulation).

The design tries to avoid putting too many dependencies on the class ns3::Node, ns3::Application, or ns3::NetDevice for the following:

- IP version, or whether IP is at all even used in the ns3::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 ns3::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.

## 16.3 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.

### 16.3.1 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).

ns3::Socket is defined in src/network/model/socket.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 peculiar 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).

### Basic operation and calls

![Implementation overview of native sockets API](image)

#### Creating sockets

An application that wants to use sockets must first create one. On real systems using a C-based API, this is accomplished by calling `socket()`

```c
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:

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

Examples of TypeIds to pass to this method are `ns3::TcpSocketFactory`, `ns3::PacketSocketFactory`, and `ns3::UdpSocketFactory`.

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

```c
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 `ns3::OnOffApplication`, the function `ns3::OnOffApplication::StartApplication()` performs the socket creation, and the application holds the socket pointer.

**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 `ns3::Socket::SetSendCallback()` callback is invoked. An application can also ask the socket how much space is available by calling `ns3::Socket::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 :cpp:func:`HandleSendCallback` is called
  * :cpp:func:`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 `ns3::Socket::SetRecvCallback()` in which the socket will notify the application when (and how much) there is data to be read, and the application then calls `ns3::Socket::Recv()` to read the data until no more can be read.

**16.3.2 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 provided for legacy API reasons. When calling the raw buffer variant of `ns3::Socket::Send()`, the buffer is immediately written into a Packet and the packet variant 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 at the application layer.
- Users can exploit the copy-on-write implementation to avoid memory copies (on the receive side, the conversion back to `uint8_t* buf` may sometimes incur an additional copy).
- Use of Packet is more aligned with the rest of the ns-3 API.

### 16.3.3 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 (zeroed) buffer on the spot, and the efficiency will be lost there.

### 16.3.4 Socket options

`to be completed`

### 16.3.5 Socket errno

`to be completed`

### 16.3.6 Example programs

`to be completed`

### 16.3.7 POSIX-like sockets API

### 16.4 Simple NetDevice

`Placeholder chapter`
CHAPTER
SEVENTEEN

OPTIMIZED LINK STATE ROUTING
(OLSR)

This model implements the base specification of the Optimized Link State Routing (OLSR) protocol, which is a
dynamic mobile ad hoc unicast routing protocol. It has been developed at the University of Murcia (Spain) by Francisco
J. Ros for NS-2, and was ported to NS-3 by Gustavo Carneiro at INESC Porto (Portugal).

17.1 Model Description

The source code for the OLSR model lives in the directory src/olsr.

17.1.1 Design

17.1.2 Scope and Limitations

The model is for IPv4 only.

- Mostly compliant with OLSR as documented in [rfc3626],
- The use of multiple interfaces was not supported by the NS-2 version, but is supported in NS-3;
- OLSR does not respond to the routing event notifications corresponding to dy-
namic interface up and down (ns3::RoutingProtocol::NotifyInterfaceUp
and ns3::RoutingProtocol::NotifyInterfaceDown) or address in-
sertion/removal (ns3::RoutingProtocol::NotifyAddAddress
and ns3::RoutingProtocol::NotifyRemoveAddress).
- Unlike the NS-2 version, does not yet support MAC layer feedback as described in RFC 3626;

Host Network Association (HNA) is supported in this implementation of OLSR. Refer to
examples/olsr-hna.cc to see how the API is used.
17.2 Usage

17.2.1 Examples

A helper class for OLSR has been written. After an IPv4 topology has been created and unique IP addresses assigned to each node, the simulation script writer can call one of three overloaded functions with different scope to enable OLSR:

- `ns3::OlsrHelper::Install (NodeContainer container);`
- `ns3::OlsrHelper::Install (Ptr<Node> node);` or
- `ns3::OlsrHelper::InstallAll (void)`

17.2.2 Helpers

In addition, the behavior of OLSR can be modified by changing certain attributes. The method `ns3::OlsrHelper::Set()` can be used to set OLSR attributes. These include HelloInterval, TcInterval, Mid-Interval, Willingness. Other parameters are defined as macros in `olsr-routing-protocol.cc`.

17.2.3 Attributes

17.2.4 Tracing

17.2.5 Logging

17.2.6 Caveats

17.3 Validation

17.3.1 Unit tests

17.3.2 Larger-scale performance tests
ns-3 simulations can use OpenFlow switches (McKeown et al. ¹), widely used in research. OpenFlow switches are configurable via the OpenFlow API, and also have an MPLS extension for quality-of-service and service-level-agreement support. By extending these capabilities to ns-3 for a simulated OpenFlow switch that is both configurable and can use the MPLS extension, ns-3 simulations can accurately simulate many different switches.

The OpenFlow software implementation distribution is hereby referred to as the OFSID. This is a demonstration of running OpenFlow in software that the OpenFlow research group has made available. There is also an OFSID that Ericsson researchers created to add MPLS capabilities; this is the OFSID currently used with ns-3. The design will allow the users to, with minimal effort, switch in a different OFSID that may include more efficient code than a previous OFSID.

18.1 Model Description

The model relies on building an external OpenFlow switch library (OFSID), and then building some ns-3 wrappers that call out to the library. The source code for the ns-3 wrappers lives in the directory `src/openflow/model`.

18.1.1 Design

The OpenFlow module presents a OpenFlowSwitchNetDevice and a OpenFlowSwitchHelper for installing it on nodes. Like the Bridge module, it takes a collection of NetDevices to set up as ports, and it acts as the intermediary between them, receiving a packet on one port and forwarding it on another, or all but the received port when flooding. Like an OpenFlow switch, it maintains a configurable flow table that can match packets by their headers and do different actions with the packet based on how it matches. The module’s understanding of OpenFlow configuration messages are kept the same format as a real OpenFlow-compatible switch, so users testing Controllers via ns-3 won’t have to rewrite their Controller to work on real OpenFlow-compatible switches.

The ns-3 OpenFlow switch device models an OpenFlow-enabled switch. It is designed to express basic use of the OpenFlow protocol, with the maintaining of a virtual Flow Table and TCAM to provide OpenFlow-like results.

The functionality comes down to the Controllers, which send messages to the switch that configure its flows, producing different effects. Controllers can be added by the user, under the ofi namespace extending ofi::Controller. To demonstrate this, a DropController, which creates flows for ignoring every single packet, and LearningController, which effectively makes the switch a more complicated BridgeNetDevice. A user versed in a standard OFSID, and/or OF protocol, can write virtual controllers to create switches of all kinds of types.

OpenFlow switch Model

The OpenFlow switch device behaves somewhat according to the diagram setup as a classical OFSID switch, with a few modifications made for a proper simulation environment.

Normal OF-enabled Switch::

| Secure Channel | <--OF Protocol--> | Controller is external |
| Hardware or Software Flow Table |

ns-3 OF-enabled Switch (module)::

| m_controller->ReceiveFromSwitch() | <--OF Protocol--> | Controller is internal |
| Software Flow Table, virtual TCAM |

In essence, there are two differences:

1) No SSL, Embedded Controller: Instead of a secure channel and connecting to an outside location for the Controller program/machine, we currently only allow a Controller extended from of::Controller, an extension of an ns3::Object. This means ns-3 programmers cannot model the SSL part of the interface or possibility of network failure. The connection to the OpenFlowSwitch is local and there aren’t any reasons for the channel/connection to break down. <<This difference may be an option in the future. Using EmuNetDevices, it should be possible to engage an external Controller program/machine, and thus work with controllers designed outside of the ns-3 environment, that simply use the proper OF protocol when communicating messages to the switch through a tap device.>>

2) Virtual Flow Table, TCAM: Typical OF-enabled switches are implemented on a hardware TCAM. The OFSID we turn into a library includes a modelled software TCAM, that produces the same results as a hardware TCAM. We include an attribute FlowTableLookupDelay, which allows a simple delay of using the TCAM to be modelled. We don’t endeavor to make this delay more complicated, based on the tasks we are running on the TCAM, that is a possible future improvement.

The OpenFlowSwitch network device is aimed to model an OpenFlow switch, with a TCAM and a connection to a controller program. With some tweaking, it can model every switch type, per OpenFlow’s extensibility. It outsources the complexity of the switch ports to NetDevices of the user’s choosing. It should be noted that these NetDevices must behave like practical switch ports, i.e. a Mac Address is assigned, and nothing more. It also must support a SendFrom function so that the OpenFlowSwitch can forward across that port.

18.1.2 Scope and Limitations

All MPLS capabilities are implemented on the OFSID side in the OpenFlowSwitchNetDevice, but ns-3-mpls hasn’t been integrated, so ns-3 has no way to pass in proper MPLS packets to the OpenFlowSwitch. If it did, one would only need to make BufferFromPacket pick up the MplsLabelStack or whatever the MPLS header is called on the Packet, and build the MPLS header into the ofpbuf.

18.1.3 Future Work

18.1.4 References

18.2 Usage

The OFSID requires libxml2 (for MPLS FIB xml file parsing), libdl (for address fault checking), and boost (for assert) libraries to be installed.
18.2.1 Building OFSID

In order to use the OpenFlowSwitch module, you must create and link the OFSID (OpenFlow Software Implementation Distribution) to ns-3. To do this:

#1 Obtain the OFSID code. An ns-3 specific OFSID branch is provided to ensure operation with ns-3. Use mercurial to download this branch and waf to build the library::

$ hg clone http://code.nsnam.org/jpelkey3/openflow
$ cd openflow

From the “openflow” directory, run::

$ ./waf configure
$ ./waf build

#2 Your OFSID is now built into a libopenflow.a library! To link to an ns-3 build with this OpenFlow switch module, run from the ns-3-dev (or whatever you have named your distribution)::

$ ./waf configure --enable-examples --enable-tests --with-openflow=path/to/openflow

#3 Under ---- Summary of optional NS-3 features: you should see::

"NS-3 OpenFlow Integration : enabled"

indicating the library has been linked to ns-3. Run::

$ ./waf build

to build ns-3 and activate the OpenFlowSwitch module in ns-3.

18.2.2 Examples

For an example demonstrating its use in a simple learning controller/switch, run::

$ ./waf --run openflow-switch

To see it in detailed logging, run::

$ ./waf --run "openflow-switch -v"

18.2.3 Helpers

18.2.4 Attributes

The SwitchNetDevice provides following Attributes:

- **FlowTableLookUpDelay**: This time gets run off the clock when making a lookup in our Flow Table.

- **Flags**: OpenFlow specific configuration flags. They are defined in the ofp_config_flags enum. Choices include:
  
  - OFPC_SEND_FLOW_EXP (Switch notifies controller when a flow has expired),
  - OFPC_FRAG_NORMAL (Match fragment against Flow table),
  - OFPC_FRAG_DROP (Drop fragments),
  - OFPC_FRAG_REASM (Reassemble only if OFPC_IP_REASM set, which is currently impossible, because switch implementation does not support IP reassembly)
  - OFPC_FRAG_MASK (Mask Fragments)

- **FlowTableMissSendLength**: When the packet doesn’t match in our Flow Table, and we forward to the controller, this sets # of bytes forwarded (packet is not forwarded in its entirety, unless specified).
18.2.5 Tracing

Note: TODO

18.2.6 Logging

Note: TODO

18.2.7 Caveats

Note: TODO

18.3 Validation

This model has one test suite which can be run as follows:

$ ./test.py --suite=openflow
This is the introduction to PointToPoint NetDevice chapter, to complement the PointToPoint model doxygen.

19.1 Overview of the PointToPoint 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 transmit 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.
19.2 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.

19.3 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 similar 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::

```cpp
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-multipoint) there may only be two nodes with associated net devices connected by a PointToPointChannel:

```cpp
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::

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

19.4 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.

19.4.1 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.

19.4.2 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.
The *ns-3* propagation module defines two generic interfaces, namely \texttt{PropagationLossModel} and \texttt{PropagationDelayModel}, for the modeling of respectively propagation loss and propagation delay.

The following propagation loss models are implemented:

- \texttt{JakesPropagationLossModel}
- \texttt{PropagationLossModel}
- \texttt{RandomPropagationLossModel}
- \texttt{FriisPropagationLossModel}
- \texttt{TwoRayGroundPropagationLossModel}
- \texttt{LogDistancePropagationLossModel}
- \texttt{ThreeLogDistancePropagationLossModel}
- \texttt{NakagamiPropagationLossModel}
- \texttt{FixedRssLossModel}
- \texttt{MatrixPropagationLossModel}
- \texttt{RangePropagationLossModel}

The following propagation delay models are implemented:

- \texttt{PropagationDelayModel}
- \texttt{RandomPropagationDelayModel}
- \texttt{ConstantSpeedPropagationDelayModel}

All the above models can be used within any model using the \texttt{PropagationLossModel} and \texttt{PropagationDelayModel} interfaces, such as Wifi and Wimax.
CHAPTER TWENTYONE

SIMPLE NETDEVICE

Placeholder chapter
This chapter outlines work on simulation data collection and the statistical framework for ns-3. The source code for the statistical framework lives in the directory src/stats.

22.1 Goals

Primary objectives for this effort are the following:

• Provide functionality to record, calculate, and present data and statistics for analysis of network simulations.
• Boost simulation performance by reducing the need to generate extensive trace logs in order to collect data.
• Enable simulation control via online statistics, e.g. terminating simulations or repeating trials.

Derived sub-goals and other target features include the following:

• Integration with the existing ns-3 tracing system as the basic instrumentation framework of the internal simulation engine, e.g. network stacks, net devices, and channels.
• Enabling users to utilize the statistics framework without requiring use of the tracing system.
• Helping users create, aggregate, and analyze data over multiple trials.
• Support for user created instrumentation, e.g. of application specific events and measures.
• Low memory and CPU overhead when the package is not in use.
• Leveraging existing analysis and output tools as much as possible. The framework may provide some basic statistics, but the focus is on collecting data and making it accessible for manipulation in established tools.
• Eventual support for distributing independent replications is important but not included in the first round of features.

22.2 Overview

The statistics framework includes the following features:

• The core framework and two basic data collectors: A counter, and a min/max/avg/total observer.
• Extensions of those to easily work with times and packets.
• Plaintext output formatted for omnetpp.
• Database output using sqlite3, a standalone, lightweight, high performance SQL engine.
• Mandatory and open ended metadata for describing and working with runs.
• An example based on the notional experiment of examining the properties of NS-3’s default ad hoc WiFi performance. It incorporates the following:
  – Constructs of a two node ad hoc WiFi network, with the nodes a parameterized distance apart.
  – UDP traffic source and sink applications with slightly different behavior and measurement hooks than the stock classes.
  – Data collection from the NS-3 core via existing trace signals, in particular data on frames transmitted and received by the WiFi MAC objects.
  – Instrumentation of custom applications by connecting new trace signals to the stat framework, as well as via direct updates. Information is recorded about total packets sent and received, bytes transmitted, and end-to-end delay.
  – An example of using packet tags to track end-to-end delay.
  – A simple control script which runs a number of trials of the experiment at varying distances and queries the resulting database to produce a graph using GNUPlot.

22.3 To-Do

High priority items include:
• Inclusion of online statistics code, e.g. for memory efficient confidence intervals.
• Provisions in the data collectors for terminating runs, i.e. when a threshold or confidence is met.
• Data collectors for logging samples over time, and output to the various formats.
• Demonstrate writing simple cyclic event glue to regularly poll some value.

Each of those should prove straightforward to incorporate in the current framework.

22.4 Approach

The framework is based around the following core principles:
• One experiment trial is conducted by one instance of a simulation program, whether in parallel or serially.
• A control script executes instances of the simulation, varying parameters as necessary.
• Data is collected and stored for plotting and analysis using external scripts and existing tools.
• Measures within the ns-3 core are taken by connecting the stat framework to existing trace signals.
• Trace signals or direct manipulation of the framework may be used to instrument custom simulation code.

Those basic components of the framework and their interactions are depicted in the following figure.
22.5 Example

This section goes through the process of constructing an experiment in the framework and producing data for analysis (graphs) from it, demonstrating the structure and API along the way.

22.5.1 Question

"What is the (simulated) performance of ns-3's WiFi NetDevices (using the default settings)? How far apart can wireless nodes be in a simulation before they cannot communicate reliably?"

- Hypothesis: Based on knowledge of real life performance, the nodes should communicate reasonably well to at least 100m apart. Communication beyond 200m shouldn't be feasible.

Although not a very common question in simulation contexts, this is an important property of which simulation developers should have a basic understanding. It is also a common study done on live hardware.

22.5.2 Simulation Program

The first thing to do in implementing this experiment is developing the simulation program. The code for this example can be found in examples/stats/wifi-example-sim.cc. It does the following main steps.

- Declaring parameters and parsing the command line using ns3::CommandLine.

```cpp
CommandLine cmd;
cmd.AddValue("distance", "Distance apart to place nodes (in meters): ", distance);
cmd.AddValue("format", "Format to use for data output.", format);
cmd.AddValue("experiment", "Identifier for experiment.",
```
• Creating nodes and network stacks using ns3::NodeContainer, ns3::WiFiHelper, and ns3::InternetStackHelper.

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

```cpp
WifiHelper wifi;
wifi.SetMac("ns3::AdhocWifiMac");
wifi.SetPhy("ns3::WifiPhy");
NetDeviceContainer nodeDevices = wifi.Install(nodes);
```

```cpp
InternetStackHelper internet;
internet.Install(nodes);
Ipv4AddressHelper ipAddrs;
ipAddrs.SetBase("192.168.0.0", "255.255.255.0");
ipAddrs.Assign(nodeDevices);
```

• Positioning the nodes using ns3::MobilityHelper. By default the nodes have static mobility and won’t move, but must be positioned the given distance apart. There are several ways to do this; it is done here using ns3::ListPositionAllocator, which draws positions from a given list.

```cpp
MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc =
    CreateObject<ListPositionAllocator>();
positionAlloc->Add(Vector(0.0, 0.0, 0.0));
positionAlloc->Add(Vector(0.0, distance, 0.0));
mobility.SetPositionAllocator(positionAlloc);
mobility.Install(nodes);
```

• Installing a traffic generator and a traffic sink. The stock Applications could be used, but the example includes custom objects in src/test/test02-apps.(cc|h). These have a simple behavior, generating a given number of packets spaced at a given interval. As there is only one of each they are installed manually; for a larger set the ns3::ApplicationHelper class could be used. The commented-out Config::Set line changes the destination of the packets, set to broadcast by default in this example. Note that in general WiFi may have different performance for broadcast and unicast frames due to different rate control and MAC retransmission policies.

```cpp
Ptr<Node> appSource = NodeList::GetNode(0);
Ptr<Sender> sender = CreateObject<Sender>();
appSource->AddApplication(sender);
sender->Start(Seconds(1));
```

```cpp
Ptr<Node> appSink = NodeList::GetNode(1);
Ptr<Receiver> receiver = CreateObject<Receiver>();
appSink->AddApplication(receiver);
receiver->Start(Seconds(0));
```

```cpp
// Config::Set("/NodeList/*/ApplicationList/*/SenderId/Destination",
//              Ipv4AddressValue("192.168.0.2");
```

• Configuring the data and statistics to be collected. The basic paradigm is that an ns3::DataCollector object is created to hold information about this particular run, to which observers and calculators are attached to

```cpp
 experiment);
cmd.AddValue("strategy", "Identifier for strategy.",
    strategy);
cmd.AddValue("run", "Identifier for run.",
    runID);
cmd.Parse (argc, argv);
```
actually generate data. Importantly, run information includes labels for the “experiment”, “strategy”, “input”,
and “run”. These are used to later identify and easily group data from multiple trials.

- The experiment is the study of which this trial is a member. Here it is on WiFi performance and distance.
- The strategy is the code or parameters being examined in this trial. In this example it is fixed, but an
  obvious extension would be to investigate different WiFi bit rates, each of which would be a different
  strategy.
- The input is the particular problem given to this trial. Here it is simply the distance between the two nodes.
- The runID is a unique identifier for this trial with which it’s information is tagged for identification in later
  analysis. If no run ID is given the example program makes a (weak) run ID using the current time.

Those four pieces of metadata are required, but more may be desired. They may be added to the record using
the ns3::DataCollector::AddMetadata() method.

```cpp
DataCollector data;
data.DescribeRun(experiment,
               strategy,
               input,
               runID);
data.AddMetadata("author", "tjkopena");
```

Actual observation and calculating is done by ns3::DataCalculator objects, of which several different
type exist. These are created by the simulation program, attached to reporting or sampling code, and then
registered with the ns3::DataCollector so they will be queried later for their output. One easy observation
mechanism is to use existing trace sources, for example to instrument objects in the ns-3 core without changing
their code. Here a counter is attached directly to a trace signal in the WiFi MAC layer on the target node.

```cpp
Ptr<PacketCounterCalculator> totalRx =
    CreateObject<PacketCounterCalculator>();
totalRx->SetKey("wifi-rx-frames");
Config::Connect("/NodeList/1/DeviceList/*/$ns3::WifiNetDevice/Rx",
    MakeCallback(&PacketCounterCalculator::FrameUpdate,
                  totalRx));
data.AddDataCalculator(totalRx);
```

Calculators may also be manipulated directly. In this example, a counter is created and passed to the traffic sink
application to be updated when packets are received.

```cpp
Ptr<CounterCalculator<> > appRx =
    CreateObject<CounterCalculator<> >();
appRx->SetKey("receiver-rx-packets");
receiver->SetCounter(appRx);
data.AddDataCalculator(appRx);
```

To increment the count, the sink’s packet processing code then calls one of the calculator’s update methods.

```cpp
m_calc->Update();
```

The program includes several other examples as well, using both the primitive calculators such as
ns3::CounterCalculator and those adapted for observing packets and times. In
src/test/test02-apps.(cc|h) it also creates a simple custom tag which it uses to track end-
to-end delay for generated packets, reporting results to a ns3::TimeMinMaxAvgTotalCalculator data
calculator.

- Running the simulation, which is very straightforward once constructed.

```cpp
Simulator::Run();
```
• Generating either omnetpp or sqlite output, depending on the command line arguments. To do this a `ns3::DataOutputInterface` object is created and configured. The specific type of this will determine the output format. This object is then given the `ns3::DataCollector` object which it interrogates to produce the output.

```cpp
Ptr<DataOutputInterface> output;
if (format == "omnet") {
    NS_LOG_INFO("Creating omnet formatted data output.");
    output = CreateObject<OmnetDataOutput>();
} else {
    #ifdef STAT_USE_DB
    NS_LOG_INFO("Creating sqlite formatted data output.");
    output = CreateObject<SqliteDataOutput>();
    #endif
}
output->Output(data);
```

• Freeing any memory used by the simulation. This should come at the end of the main function for the example.

```cpp
Simulator::Destroy();
```

### Logging

To see what the example program, applications, and stat framework are doing in detail, set the `NS_LOG` variable appropriately. The following will provide copious output from all three.

```bash
export NS_LOG=StatFramework:WiFiDistanceExperiment:WiFiDistanceApps
```

Note that this slows down the simulation extraordinarily.

### Sample Output

Compiling and simply running the test program will append omnet++ formatted output such as the following to `data.sca`.

```bash
run run-1212239121
attr experiment "wifi-distance-test"
attr strategy "wifi-default"
attr input "50"
attr description ""
attr "author" "tjkopena"
scalar wifi-tx-frames count 30
scalar wifi-rx-frames count 30
scalar sender-tx-packets count 30
scalar receiver-rx-packets count 30
scalar tx-pkt-size count 30
scalar tx-pkt-size total 1920
scalar tx-pkt-size average 64
scalar tx-pkt-size max 64
scalar tx-pkt-size min 64
scalar delay count 30
scalar delay total 5884980ns
scalar delay average 196166ns
```
22.5.3 Control Script

In order to automate data collection at a variety of inputs (distances), a simple Bash script is used to execute a series of simulations. It can be found at examples/stats/wifi-example-db.sh. The script runs through a set of distances, collecting the results into an sqlite3 database. At each distance five trials are conducted to give a better picture of expected performance. The entire experiment takes only a few dozen seconds to run on a low end machine as there is no output during the simulation and little traffic is generated.

```
#!/bin/sh

DISTANCES="25 50 75 100 125 145 150 152 155 157 160 162 165 167 170 172 175 177 180"
TRIALS="1 2 3 4 5"

echo WiFi Experiment Example

if [ -e data.db ]
then
  echo Kill data.db?
  read ANS
  if [ "$ANS" = "yes" -o "$ANS" = "y" ]
  then
    echo Deleting database
    rm data.db
  fi
fi

for trial in $TRIALS
do
  for distance in $DISTANCES
  do
    echo Trial $trial, distance $distance
    ./bin/test02 --format=db --distance=$distance --run=run-$distance-$trial
  done
done

```

22.5.4 Analysis and Conclusion

Once all trials have been conducted, the script executes a simple SQL query over the database using the sqlite3 command line program. The query computes average packet loss in each set of trials associated with each distance. It does not take into account different strategies, but the information is present in the database to make some simple extensions and do so. The collected data is then passed to GNUPlot for graphing.

```
CMD="select exp.input,avg(100-((rx.value*100)/tx.value)) \n  from Singletons rx, Singletons tx, Experiments exp \n  where rx.run = tx.run AND \n    rx.run = exp.run AND \n    rx.name=’receiver-rx-packets’ AND \n    tx.name=’sender-tx-packets’ \n  group by exp.input \n  order by abs(exp.input) ASC;"

sqlite3 -noheader data.db "$CMD" > wifi-default.data
```

22.5. Example
The GNUPlot script found at examples/stats/wifi-example.gnuplot simply defines the output format and some basic formatting for the graph.

```
set terminal postscript portrait enhanced lw 2 "Helvetica" 14
set size 1.0, 0.66
#-------------------------------------------------------
set out "wifi-default.eps"
#set title "Packet Loss Over Distance"
set xlabel "Distance (m) --- average of 5 trials per point"
set xrange [0:200]
set ylabel "% Packet Loss"
set yrange [0:110]
plot "wifi-default.data" with lines title "WiFi Defaults"
```

### End Result

The resulting graph provides no evidence that the default WiFi model’s performance is necessarily unreasonable and lends some confidence to an at least token faithfulness to reality. More importantly, this simple investigation has been carried all the way through using the statistical framework. Success!
% Packet Loss --- average of 5 trials per distance

Distance (m)
The main goal of the UAN Framework is to enable researchers to model a variety of underwater network scenarios. The UAN model is broken into four main parts: The channel, PHY, MAC and Autonomous Underwater Vehicle (AUV) models.

The need for underwater wireless communications exists in applications such as remote control in offshore oil industry, pollution monitoring in environmental systems, speech transmission between divers, mapping of the ocean floor, mine counter measures, seismic monitoring of ocean faults as well as climate changes monitoring. Unfortunately, making on-field measurements is very expensive and there are no commonly accepted standard to base on. Hence, the priority to make research work going on, it is to realize a complete simulation framework that researchers can use to experiment, make tests and make performance evaluation and comparison.

The NS-3 UAN module is a first step in this direction, trying to offer a reliable and realistic tool. In fact, the UAN module offers accurate modelling of the underwater acoustic channel, a model of the WHOI acoustic modem (one of the widely used acoustic modems) and its communications performance, and some MAC protocols.

23.1 Model Description

The source code for the UAN Framework lives in the directory src/uan and in src/energy for the contribution on the li-ion battery model.

The UAN Framework is composed of two main parts:

- the AUV mobility models, including Electric motor propelled AUV (REMUS class) and Seaglider models
- the energy models, including AUV energy models, AUV energy sources (batteries) and an acoustic modem energy model

As enabling component for the energy models, a Li-Ion batteries energy source has been implemented basing on

---

3 Hydroinc Products; URL: http://www.hydroidinc.com/products.html
23.1.1 Design

UAN Propagation Models

Modelling of the underwater acoustic channel has been an active area of research for quite some time. Given the complications involved, surface and bottom interactions, varying speed of sound, etc..., the detailed models in use for ocean acoustics research are much too complex (in terms of runtime) for use in network level simulations. We have attempted to provide the often used models as well as make an attempt to bridge, in part, the gap between complicated ocean acoustic models and network level simulation. The three propagation models included are the ideal channel model, the Thorp propagation model and the Bellhop propagation model (Available as an addition).

All of the Propagation Models follow the same simple interface in ns3::UanPropModel. The propagation models provide a power delay profile (PDP) and pathloss information. The PDP is retrieved using the GetPdp method which returns type UanPdp. ns3::UanPdp utilises a tapped delay line model for the acoustic channel. The UanPdp class is a container class for Taps, each tap has a delay and amplitude member corresponding to the time of arrival (relative to the first tap arrival time) and amplitude. The propagation model also provides pathloss between the source and receiver in dB re 1uPa. The PDP and pathloss can then be used to find the received signal power over a duration of time (i.e. received signal power in a symbol duration and ISI which interferes with neighbouring signals). Both UanPropModelIdeal and UanPropModelThorp return a single impulse for a PDP.

1. Ideal Channel Model ns3::UanPropModelIdeal

The ideal channel model assumes 0 pathloss inside a cylindrical area with bounds set by attribute. The ideal channel model also assumes an impulse PDP.

2. Thorp Propagation Model ns3::UanPropModelThorp

The Thorp Propagation Model calculates pathloss using the well-known Thorp approximation. This model is similar to the underwater channel model implemented in ns2 as described here:


The frequency used in calculation however, is the center frequency of the modulation as found from ns3::UanTxMode. The Thorp Propagation Model also assumes an impulse channel response.

3. Bellhop Propagation Model ns3::UanPropModelBh (Available as an addition)

The Bellhop propagation model reads propagation information from a database. A configuration file describing the location, and resolution of the archived information must be supplied via attributes. We have included a utility, create-dat, which can create these data files using the Bellhop Acoustic Ray Tracing software (http://oalib.hlsresearch.com/).

The create-dat utility requires a Bellhop installation to run. Bellhop takes environment information about the channel, such as sound speed profile, surface height bottom type, water depth, and uses a gaussian ray tracing algorithm to determine propagation information. Arrivals from Bellhop are grouped together into equal length taps (the arrivals in a tap duration are coherently summed). The maximum taps are then aligned to take the same position in the PDP. The create-dat utility averages together several runs and then normalizes the average such that the sum of all taps is 1. The same configuration file used to create the data files using create-dat should be passed via attribute to the Bellhop Propagation Model.

The Bellhop propagation model is available as a patch. The link address will be made available here when it is posted online. Otherwise email lentracy@gmail.com for more information.

UAN PHY Model Overview

The PHY has been designed to allow for relatively easy extension to new networking scenarios. We feel this is important as, to date, there has been no commonly accepted network level simulation model for underwater networks.
The lack of commonly accepted network simulation tools has resulted in a wide array of simulators and models used to report results in literature. The lack of standardization makes comparing results nearly impossible.

The main component of the PHY Model is the generic PHY class, ns3::UanPhyGen. The PHY class's general responsibility is to handle packet acquisition, error determination, and forwarding of successful packets up to the MAC layer. The Generic PHY uses two models for determination of signal to noise ratio (SINR) and packet error rate (PER). The combination of the PER and SINR models determine successful reception of packets. The PHY model connects to the channel via a Transducer class. The Transducer class is responsible for tracking all arriving packets and departing packets over the duration of the events. How the PHY class and the PER and SINR models respond to packets is based on the “Mode” of the transmission as described by the ns3::UanTxMode class.

When a MAC layer sends down a packet to the PHY for transmission it specifies a “mode number” to be used for the transmission. The PHY class accepts, as an attribute, a list of supported modes. The mode number corresponds to an index in the supported modes. The UanTxMode contains simple modulation information and a unique string id. The generic PHY class will only acquire arriving packets which use a mode which is in the supported modes list of the PHY. The mode along with received signal power, and other pertinent attributes (e.g. possibly interfering packets and their modes) are passed to the SINR and PER models for calculation of SINR and probability of error.

Several simple example PER and SINR models have been created. a) The PER models - Default (simple) PER model (ns3::UanPhyPerGenDefault): The Default PER model tests the packet against a threshold and assumes error (with prob. 1) if the SINR is below the threshold or success if the SINR is above the threshold - Micromodem FH-FSK PER (ns3::UanPhyPerUmodem). The FH-FSK PER model calculates probability of error assuming a rate 1/2 convolutional code with constraint length 9 and a CRC check capable of correcting up to 1 bit error. This is similar to what is used in the receiver of the WHOI Micromodem.

b) SINR models - Default Model (ns3::UanPhyCalcSinrDefault). The default SINR model assumes that all transmitted energy is captured at the receiver and that there is no ISI. Any received signal power from interferes acts as additional ambient noise. - FH-FSK SINR Model (ns3::UanPhyCalcSinrFhFsk). The WHOI Micromodem operating in FH-FSK mode uses a predetermined hopping pattern that is shared by all nodes in the network. We model this by only including signal energy receiving within one symbol time (as given by ns3::UanTxMode) in calculating the received signal power. A channel clearing time is given to the FH-FSK SINR model via attribute. Any signal energy arriving in adjacent signals (after a symbol time and the clearing time) is considered ISI and is treated as additional ambient noise. Interfering signal arrivals inside a symbol time (any symbol time) is also counted as additional ambient noise - Frequency filtered SINR (ns3::UanPhyCalcSinrDual). This SINR model calculates SINR in the same manner as the default model. This model however only considers interference if there is an overlap in frequency of the arriving packets as determined by UanTxMode.

In addition to the generic PHY a dual phy layer is also included (ns3::UanPhyDual). This wraps two generic phy layers together to model a net device which includes two receivers. This was primarily developed for UanMacRc, described in the next section.

**UAN MAC Model Overview**

Over the last several years there have been a myriad of underwater MAC proposals in the literature. We have included three MAC protocols with this distribution: a) CW-MAC, a MAC protocol which uses a slotted contention window similar in nature to the IEEE 802.11 DCF. Nodes have a constant contention window measured in slot times (configured via attribute). If the channel is sensed busy, then nodes backoff by randomly (uniform distribution) choose a slot to transmit in. The slot time durations are also configured via attribute. This MAC was described in Parrish N.; Tracy L.; Roy S. Arabshahi P.; and Fox, W., System Design Considerations for Undersea Networks: Link and Multiple Access Protocols , IEEE Journal on Selected Areas in Communications (JSAC), Special Issue on Underwater Wireless Communications and Networks, Dec. 2008.

b) RC-MAC (ns3::UanMacRc ns3::UanMacRcGw) a reservation channel protocol which dynamically divides the available bandwidth into a data channel and a control channel. This MAC protocol assumes there is a gateway node which all network traffic is destined for. The current implementation assumes a single gateway and a single network neighborhood (a single hop network). RTS/CTS handshaking is used and time is divided into cycles. Non-gateway
nodes transmit RTS packets on the control channel in parallel to data packet transmissions which were scheduled in the previous cycle at the start of a new cycle, the gateway responds on the data channel with a CTS packet which includes packet transmission times of data packets for received RTS packets in the previous cycle as well as bandwidth allocation information. At the end of a cycle ACK packets are transmitted for received data packets.

When a publication is available it will be cited here.

3. Simple ALOHA (ns3::UanMacAloha) Nodes transmit at will.

AUV mobility models

The AUV mobility models have been designed as in the follows.

Use cases

The user will be able to:

- program the AUV to navigate over a path of waypoints
- control the velocity of the AUV
- control the depth of the AUV
- control the direction of the AUV
- control the pitch of the AUV
- tell the AUV to emerge or submerge to a specified depth

AUV mobility models design

Implement a model of the navigation of AUV. This involves implementing two classes modelling the two major categories of AUVs: electric motor propelled (like REMUS class \(^3\) \(^4\)) and “sea gliders” \(^5\). The classic AUVs are submarine-like devices, propelled by an electric motor linked with a propeller. Instead, the “sea glider” class exploits small changes in its buoyancy that, in conjunction with wings, can convert vertical motion to horizontal. So, a glider will reach a point into the water by describing a “saw-tooth” movement. Modelling the AUV navigation, involves in considering a real-world AUV class thus, taking into account maximum speed, directional capabilities, emerging and submerging times. Regarding the sea gliders, it is modelled the characteristic saw-tooth movement, with AUV’s speed driven by buoyancy and glide angle.

Figure 23.1: AUV’s mobility model classes overview
An `ns3::AuvMobilityModel` interface has been designed to give users a generic interface to access AUV’s navigation functions. The AuvMobilityModel interface is implemented by the RemusMobilityModel and the GliderMobilityModel classes. The AUV’s mobility models organization is shown in *AUV’s mobility model classes overview*. Both models use a constant velocity movement, thus the AuvMobilityModel interface derives from the ConstantVelocityMobilityModel. The two classes hold the navigation parameters for the two different AUVs, like maximum pitch angles, maximum operating depth, maximum and minimum speed values. The Glider model holds also some extra parameters like maximum buoyancy values, and maximum and minimum glide slopes. Both classes, RemusMobilityModel and GliderMobilityModel, handle also the AUV power consumption, utilizing the relative power models. Has been modified the WaypointMobilityModel to let it use a generic underlying ConstantVelocityModel to validate the waypoints and, to keep trace of the node’s position. The default model is the classic ConstantVelocityModel but, for example in case of REMUS mobility model, the user can install the AUV mobility model into the waypoint model and then validating the waypoints against REMUS navigation constraints.

**Energy models**

The energy models have been designed as in the follows.

**Use cases**

The user will be able to:
- use a specific power profile for the acoustic modem
- use a specific energy model for the AUV
- trace the power consumption of AUV navigation, through AUV’s energy model
- trace the power consumption underwater acoustic communications, through acoustic modem power profile

We have integrated the Energy Model with the UAN module, to implement energy handling. We have implemented a specific energy model for the two AUV classes and, an energy source for Lithium batteries. This will be really useful for researchers to keep trace of the AUV operational life. We have implemented also an acoustic modem power profile, to keep trace of its power consumption. This can be used to compare protocols specific power performance. In order to use such power profile, the acoustic transducer physical layer has been modified to use the modem power profile. We have decoupled the physical layer from the transducer specific energy model, to let the users change the different energy models without changing the physical layer.

**AUV energy models**

Basing on the Device Energy Model interface, it has been implemented a specific energy model for the two AUV classes (REMUS and Seaglider). This models reproduce the AUV’s specific power consumption to give users accurate information. This model can be naturally used to evaluate the AUV operating life, as well as mission-related power consumption, etc. Have been developed two AUV energy models:
- GliderEnergyModel, computes the power consumption of the vehicle based on the current buoyancy value and vertical speed
- RemusEnergyModel, computes the power consumption of the vehicle based on the current speed, as it is propelled by a brush-less electric motor

**Note:** TODO extend a little bit
AUV energy sources

Note: [TODO]

Acoustic modem energy model

Basing on the Device Energy Model interface, has been implemented a generic energy model for acoustic modem. The model allows to trace four modem’s power-states: Sleep, Idle, Receiving, Transmitting. The default parameters for the energy model are set to fit those of the WHOI \( \mu \text{modem} \). The class follows pretty closely the RadioEnergyModel class as the transducer behaviour is pretty close to that of a wifi radio.

The default power consumption values implemented into the model are as follows [6]:

<table>
<thead>
<tr>
<th>Modem State</th>
<th>Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX</td>
<td>50 W</td>
</tr>
<tr>
<td>RX</td>
<td>158 mW</td>
</tr>
<tr>
<td>Idle</td>
<td>158 mW</td>
</tr>
<tr>
<td>Sleep</td>
<td>5.8 mW</td>
</tr>
</tbody>
</table>

UAN module energy modifications

The UAN module has been modified in order to utilize the implemented energy classes. Specifically, it has been modified the physical layer of the UAN module. It has been implemented an UpdatePowerConsumption method that takes the modem’s state as parameter. It checks if an energy source is installed into the node and, in case, it then use the AcousticModemEnergyModel to update the power consumption with the current modem’s state. The modem power consumption’s update takes place whenever the modem changes its state.

A user should take into account that, if the the power consumption handling is enabled (if the node has an energy source installed), all the communications processes will terminate whether the node depletes all the energy source.

Li-Ion batteries model

A generic Li-Ion battery model has been implemented based on [7][8]. The model can be fitted to any type of Li-Ion battery simply changing the model’s parameters. The default values are fitted for the Panasonic CGR18650DA Li-Ion Battery [9]. [TODO insert figure] As shown in figure the model approximates very well the Li-Ion cells. Regarding Seagliders, the batteries used into the AUV are Electrochem 3B36 Lithium / Sulfuryl Chloride cells [10]. Also with this cell type, the model seems to approximates the different discharge curves pretty well, as shown in the figure.

Note: should I insert the li-ion model deatils here? I think it is better to put them into an Energy-related chapter..

23.1.2 Scope and Limitations

The framework is designed to simulate AUV’s behaviour. We have modeled the navigation and power consumption behaviour of REMUS class and Seaglider AUVs. The communications stack, associated with the AUV, can be modified depending on simulation needs. Usually, the default underwater stack is being used, composed of an half duplex acoustic modem, an Aloha MAC protocol and a generic physical layer.

Regarding the AUV energy consumption, the user should be aware that the level of accuracy differs for the two classes:
• Seaglider, high level of accuracy, thanks to the availability of detailed information on AUV’s components and
behaviour [5] [10]. Have been modeled both the navigation power consumption and the Li battery packs (ac-
cording to [5]).

• REMUS, medium level of accuracy, due to the lack of publicly available information on AUV’s components.
We have approximated the power consumption of the AUV’s motor with a linear behaviour and, the energy
source uses an ideal model (BasicEnergySource) with a power capacity equal to that specified in [4].

23.1.3 Future Work

Some ideas could be:

• insert a data logging capability
• modify the framework to use sockets (enabling the possibility to use applications)
• introduce some more MAC protocols
• modify the physical layer to let it consider the doppler spread (problematic in underwater environments)
• introduce OFDM modulations

23.1.4 References

23.2 Usage

The main way that users who write simulation scripts will typically interact with the UAN Framework is through the
helper API and through the publicly visible attributes of the model.

The helper API is defined in src/uan/helper/acoustic-modem-energy-model-helper.{cc,h} and
in /src/uan/helper/...{cc,h}.

The example folder src/uan/examples/ contain some basic code that shows how to set up and use the models.
Further examples can be found into the Unit tests in src/uan/test/...cc

23.2.1 Examples

Examples of the Framework’s usage can be found into the examples folder. There are mobility related examples and
uan related ones.

Mobility Model Examples

• **auv-energy-model**: In this example we show the basic usage of an AUV energy model. Specifically, we
show how to create a generic node, adding to it a basic energy source and consuming energy from the
energy source. In this example we show the basic usage of an AUV energy model.

The Seaglider AUV power consumption depends on buoyancy and vertical speed values, so we simulate a
20 seconds movement at 0.3 m/s of vertical speed and 138g of buoyancy. Then a 20 seconds movement at
0.2 m/s of vertical speed and 138g of buoyancy and then a stop of 5 seconds.

The required energy will be drained by the model basing on the given buoyancy/speed values, from the
energy source installed onto the node. We finally register a callback to the TotalEnergyConsumption traced
value.
• **aau-mobility**: In this example we show how to use the AuvMobilityHelper to install an AUV mobility model into a (set of) node. Then we make the AUV to submerge to a depth of 1000 meters. We then set a callback function called on reaching of the target depth. The callback then makes the AUV to emerge to water surface (0 meters). We set also a callback function called on reaching of the target depth. The emerge callback then, stops the AUV.

During the whole navigation process, the AUV’s position is tracked by the TracePos function and plotted into a Gnuplot graph.

• **waypoint-mobility**: We show how to use the WaypointMobilityModel with a non-standard ConstantVelocityMobilityModel. We first create a waypoint model with an underlying RemusMobilityModel setting the mobility trace with two waypoints. We then create a waypoint model with an underlying GliderMobilityModel setting the waypoints separately with the AddWaypoint method. The AUV’s position is printed out every seconds.

**UAN Examples**

• **li-ion-energy-source** In this simple example, we show how to create and drain energy from a LiIonEnergySource. We make a series of discharge calls to the energy source class, with different current drain and durations, until all the energy is depleted from the cell (i.e. the voltage of the cell goes below the threshold level). Every 20 seconds we print out the actual cell voltage to verify that it follows the discharge curve [9]. At the end of the example it is verified that after the energy depletion call, the cell voltage is below the threshold voltage.

• **uan-energy-auv** This is a comprehensive example where all the project’s components are used. We setup two nodes, one fixed surface gateway equipped with an acoustic modem and a moving Seaglider AUV with an acoustic modem too. Using the waypoint mobility model with an underlying GliderMobilityModel, we make the glider descend to -1000 meters and then emerge to the water surface. The AUV sends a generic 17-bytes packet every 10 seconds during the navigation process. The gateway receives the packets and stores the total bytes amount. At the end of the simulation are shown the energy consumptions of the two nodes and the networking stats.

**23.2.2 Helpers**

In this section we give an overview of the available helpers and their behaviour.

**AcousticModemEnergyModelHelper**

This helper installs AcousticModemEnergyModel into UanNetDevice objects only. It requires an UanNetDevice and an EnergySource as input objects.

The helper creates an AcousticModemEnergyModel with default parameters and associate it with the given energy source. It configures an EnergyModelCallback and an EnergyDepletionCallback. The depletion callback can be configured as a parameter.

**AuvGliderHelper**

Installs into a node (or set of nodes) the Seaglider’s features:

• waypoint model with underlying glider mobility model

• glider energy model

• glider energy source
• micro modem energy model

The glider mobility model is the GliderMobilityModel with default parameters. The glider energy model is the GliderEnergyModel with default parameters.

Regarding the energy source, the Seaglider features two battery packs, one for motor power and one for digital-analog power. Each pack is composed of 12 (10V) and 42 (24V) lithium chloride DD-cell batteries, respectively [5]. The total power capacity is around 17.5 MJ (3.9 MJ + 13.6 MJ). In the original version of the Seaglider there was 18 + 63 D-cell with a total power capacity of 10MJ.

The packs design is as follows:

- 10V - 3 in-series string x 4 strings = 12 cells - typical capacity ~100 Ah
- 24V - 7 in-series-strings x 6 strings = 42 cells - typical capacity ~150 Ah

Battery cells are Electrochem 3B36, with 3.6 V nominal voltage and 30.0 Ah nominal capacity. The 10V battery pack is associated with the electronic devices, while the 24V one is associated with the pump motor.

The micro modem energy model is the MicroModemEnergyModel with default parameters.

---

**AuvRemusHelper**

Install into a node (or set of nodes) the REMUS features:

- waypoint model with REMUS mobility model validation
- REMUS energy model
- REMUS energy source
- micro modem energy model

The REMUS mobility model is the RemusMobilityModel with default parameters. The REMUS energy model is the RemusEnergyModel with default parameters.

Regarding the energy source, the REMUS features a rechargeable lithium ion battery pack rated 1.1 kWh @ 27 V (40 Ah) in operating conditions (specifications from [3] and Hydroinc European salesman). Since more detailed information about battery pack were not publicly available, the energy source used is a BasicEnergySource.

The micro modem energy model is the MicroModemEnergyModel with default parameters.

---

### 23.2.3 Attributes

**Note:** TODO

### 23.2.4 Tracing

**Note:** TODO

### 23.2.5 Logging

**Note:** TODO
23.2.6 Caveats

Note: TODO

23.3 Validation

This model has been tested with three UNIT test:

- auv-energy-model
- auv-mobility
- li-ion-energy-source

23.3.1 Auv Energy Model

Includes test cases for single packet energy consumption, energy depletion, Glider and REMUS energy consumption. The unit test can be found in src/uan/test/auv-energy-model-test.cc.

The single packet energy consumption test do the following:

- creates a two node network, one surface gateway and one fixed node at -500 m of depth
- install the acoustic communication stack with energy consumption support into the nodes
- a packet is sent from the underwater node to the gateway
- it is verified that both, the gateway and the fixed node, have consumed the expected amount of energy from their sources

The energy depletion test do the following steps:

- create a node with an empty energy source
- try to send a packet
- verify that the energy depletion callback has been invoked

The Glider energy consumption test do the following:

- create a node with glider capabilities
- make the vehicle to move to a predetermined waypoint
- verify that the energy consumed for the navigation is correct, according to the glider specifications

The REMUS energy consumption test do the following:

- create a node with REMUS capabilities
- make the vehicle to move to a predetermined waypoint
- verify that the energy consumed for the navigation is correct, according to the REMUS specifications

23.3.2 Auv Mobility

Includes test cases for glider and REMUS mobility models. The unit test can be found in src/uan/test/auv-mobility-test.cc.
• create a node with glider capabilities
• set a specified velocity vector and verify if the resulting buoyancy is the one that is supposed to be
• make the vehicle to submerge to a specified depth and verify if, at the end of the process the position is the one that is supposed to be
• make the vehicle to emerge to a specified depth and verify if, at the end of the process the position is the one that is supposed to be
• make the vehicle to navigate to a specified point, using direction, pitch and speed settings and, verify if at the end of the process the position is the one that is supposed to be
• make the vehicle to navigate to a specified point, using a velocity vector and, verify if at the end of the process the position is the one that is supposed to be

The REMUS mobility model test do the following:
• create a node with glider capabilities
• make the vehicle to submerge to a specified depth and verify if, at the end of the process the position is the one that is supposed to be
• make the vehicle to emerge to a specified depth and verify if, at the end of the process the position is the one that is supposed to be
• make the vehicle to navigate to a specified point, using direction, pitch and speed settings and, verify if at the end of the process the position is the one that is supposed to be
• make the vehicle to navigate to a specified point, using a velocity vector and, verify if at the end of the process the position is the one that is supposed to be

23.3.3 Li-Ion Energy Source

Includes test case for Li-Ion energy source. The unit test can be found in src/energy/test/li-ion-energy-source-test.cc.

The test case verify that after a well-known discharge time with constant current drain, the cell voltage has followed the datasheet discharge curve [9].
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.

### 24.1 Overview of the model

The WifiNetDevice models a wireless network interface controller based on the IEEE 802.11 standard [ieee80211]. We will go into more detail below but in brief, ns-3 provides models for these aspects of 802.11:

- basic 802.11 DCF with **infrastructure** and **adhoc** modes
- **802.11a**, **802.11b** and **802.11g** physical layers
- QoS-based EDCA and queueing extensions of **802.11e**
- various propagation loss models including Nakagami, Rayleigh, Friis, LogDistance, FixedRss, Random
- two propagation delay models, a distance-based and random model
- various rate control algorithms including Aarf, Arf, Cara, Onoe, Rraa, ConstantRate, and Minstrel
- **802.11s** (mesh), described in another chapter

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 implementation is modular and provides roughly four levels of models:

- the **PHY layer models**
- the so-called **MAC low models**: they implement DCF and EDCAF
- 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** that provide for the three (non-mesh; the mesh equivalent, which is a sibling of these with common parent ns3::RegularWifiMac, is not discussed here) Wi-Fi topological elements - Access Point (AP) (implemented in class ns3::ApWifiMac), non-AP Station (STA) (ns3::StaWifiMac), and STA in an Independent Basic Service Set (IBSS - also commonly referred to as an ad hoc network (ns3::AdhocWifiMac).

The simplest of these is ns3::AdhocWifiMac, which implements a Wi-Fi MAC that does not perform any kind of beacon generation, probing, or association. The ns3::StaWifiMac class implements an active probing and
ns-3 Model Library, Release ns-3-dev

association state machine that handles automatic re-association whenever too many beacons are missed. Finally, 
ns3::ApWifiMac implements an AP that generates periodic beacons, and that accepts every attempt to associate.

These three MAC high models share a common parent in ns3::RegularWifiMac, which exposes, among other MAC configuration, an attribute QosSupported that allows configuration of 802.11e/WMM-style QoS support. With QoS-enabled MAC models it is possible to work with traffic belonging to four different Access Categories (ACs):  
AC_VO for voice traffic, AC_VI for video traffic, AC_BE for best-effort traffic and AC_BK for background traffic. In order for the MAC to determine the appropriate AC for an MSDU, packets forwarded down to these MAC layers should be marked using ns3::QosTag in order to set a TID (traffic id) for that packet otherwise it will be considered belonging to AC_BE.

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 and EDCAF functions.
3. ns3::DcaTxop and ns3::EdcaTxopN which handle the packet queue, packet fragmentation, and packet retransmissions if they are needed. The ns3::DcaTxop object is used high MACs that are not QoS-enabled, and for transmission of frames (e.g., of type Management) that the standard says should access the medium using the DCF. ns3::EdcaTxopN is is used by QoS-enabled high MACs and also performs QoS operations like 802.11n-style MSDU aggregation.

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

- OnoeWifiManager
- IdealWifiManager
- AarfcdWifiManager
- AarfWifiManager
- ArfWifiManager
- AmrrWifiManager
- ConstantRateWifiManager
- MinstrelWifiManager
- CaraWifiManager
- RraaWifiManager

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 Validation results for 802.11b are available in this technical report.

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/wifi.

### 24.2 Using the WifiNetDevice

The modularity provided by the implementation makes low-level configuration of the WifiNetDevice powerful but complex. For this reason, we provide some helper classes to perform common operations in a simple matter, and leverage the ns-3 attribute system to allow users to control the parametrization of the underlying models.
24.2. Using the WifiNetDevice

Figure 24.1: Wifi NetDevice architecture.
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 constituent 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.

### 24.2.1 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:

```cpp
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*

### 24.2.2 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:

```cpp
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.

### 24.2.3 NqosWifiMacHelper and QosWifiMacHelper

The ns3::NqosWifiMacHelper and ns3::QosWifiMacHelper configure an object factory to create instances of a ns3::WifiMac. They are used to configure MAC parameters like type of MAC.

The former, ns3::NqosWifiMacHelper, supports creation of MAC instances that do not have 802.11e/WMM-style QoS support enabled.
For example the following user code configures a non-QoS MAC that will be a non-AP STA in an infrastructure network where the AP has SSID ns-3-ssid:

```c++
NqosWifiMacHelper wifiMacHelper = NqosWifiMacHelper::Default ();
Ssid ssid = Ssid ("ns-3-ssid");
wifiMacHelper.SetType ("ns3::StaWifiMac",
    "Ssid", SsidValue (ssid),
    "ActiveProbing", BooleanValue (false));
```

To create MAC instances with QoS support enabled, ns3::QosWifiMacHelper is used in place of ns3::NqosWifiMacHelper. This object can be also used to set:

- a MSDU aggregator for a particular Access Category (AC) in order to use 802.11n MSDU aggregation feature;
- block ack parameters like threshold (number of packets for which block ack mechanism should be used) and inactivity timeout.

The following code shows an example use of ns3::QosWifiMacHelper to create an AP with QoS enabled, aggregation on AC_VO, and Block Ack on AC_BE:

```c++
QosWifiMacHelper wifiMacHelper = QosWifiMacHelper::Default ();
wifiMacHelper.SetType ("ns3::ApWifiMac",
    "Ssid", SsidValue (ssid),
    "BeaconGeneration", BooleanValue (true),
    "BeaconInterval", TimeValue (Seconds (2.5)));
```

24.2.4 WifiHelper

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

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

What does this do? It sets the RemoteStationManager to ns3::ArfWifiManager. Now, let's use the wifiPhyHelper and wifiMacHelper created above to install WifiNetDevices on a set of nodes in a NodeContainer "c":

```c++
NetDeviceContainer wifiContainer = WifiHelper::Install (wifiPhyHelper, wifiMacHelper, 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.

24.2.5 AdHoc WifiNetDevice configuration

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

*To be completed*

24.2.6 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.

*To be completed*
24.3 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. For the channel propagation modeling, the propagation module is used; see section propagation for details.

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 24.2: SNIR function over time. and calculate the SNIR function.
From the SNIR function we can derive the Bit Error Rate (BER) and Packet Error Rate (PER) for the modulation and coding scheme being used for the transmission. Please refer to [pei80211ofdm], [pei80211b] and [lacage2006yans] for a detailed description of the available BER/PER models.

### 24.3.1 WifiChannel configuration

The WifiChannel implementation uses the propagation loss and delay models provided within the ns-3 *propagation* module.

### 24.4 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 [ji2004sslswn] 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,
- msdu aggregation,
- etc.

### 24.5 Wifi Attributes

Should link to the list of attributes exported by Doxygen

### 24.6 Wifi Tracing

Should link to the list of traces exported by Doxygen

### 24.7 References
This chapter describes the ns-3 WimaxNetDevice and related models. By adding WimaxNetDevice objects to ns-3 nodes, one can create models of 802.16-based networks. Below, we list some more details about what the ns-3 WiMAX models cover but, in summary, the most important features of the ns-3 model are:

- a scalable and realistic physical layer and channel model
- a packet classifier for the IP convergence sublayer
- efficient uplink and downlink schedulers
- support for Multicast and Broadcast Service (MBS), and
- packet tracing functionality

The source code for the WiMAX models lives in the directory src/wimax.

There have been two academic papers published on this model:


### 25.1 Scope of the model

From a MAC perspective, there are two basic modes of operation, that of a Subscriber Station (SS) or a Base Station (BS). These are implemented as two subclasses of the base class ns3::NetDevice, class SubscriberStationNetDevice and class BaseStationNetDevice. As is typical in ns-3, there is also a physical layer class WimaxPhy and a channel class WimaxChannel which serves to hold the references to all of the attached Phy devices. The main physical layer class is the SimpleOfdmWimaxChannel class.

Another important aspect of WiMAX is the uplink and downlink scheduler, and there are three primary scheduler types implemented:

- SIMPLE: a simple priority based FCFS scheduler
- RTPS: a real-time polling service (rtPS) scheduler
- MBQOS: a migration-based uplink scheduler

The following additional aspects of the 802.16 specifications, as well as physical layer and channel models, are modelled:

- leverages existing ns-3 wireless propagation loss and delay models, as well as ns-3 mobility models
• Point-to-Multipoint (PMP) mode and the WirelessMAN-OFDM PHY layer
• Initial Ranging
• Service Flow Initialization
• Management Connection
• Transport Initialization
• UGS, rtPS, nrtPS, and BE connections

The following aspects are not presently modelled but would be good topics for future extensions:

• OFDMA PHY layer
• Link adaptation
• Mesh topologies
• ARQ
• ertPS connection
• packet header suppression

25.2 Using the Wimax models

The main way that users who write simulation scripts will typically interact with the Wimax models is through the helper API and through the publicly visible attributes of the model.

The helper API is defined in src/wimax/helper/wimax-helper.{cc,h}.

The example src/wimax/examples/wimax-simple.cc contains some basic code that shows how to set up the model:

```
switch (schedType)
{
  case 0:
    scheduler = WimaxHelper::SCHED_TYPE_SIMPLE;
    break;
  case 1:
    scheduler = WimaxHelper::SCHED_TYPE_MBQOS;
    break;
  case 2:
    scheduler = WimaxHelper::SCHED_TYPE_RTPS;
    break;
  default:
    scheduler = WimaxHelper::SCHED_TYPE_SIMPLE;
}

NodeContainer ssNodes;
NodeContainer bsNodes;

ssNodes.Create (2);
bsNodes.Create (1);

WimaxHelper wimax;

NetDeviceContainer ssDevs, bsDevs;

ssDevs = wimax.Install (ssNodes,
This example shows that there are two subscriber stations and one base station created. The helper method `Install` allows the user to specify the scheduler type, the physical layer type, and the device type.

Different variants of `Install` are available; for instance, the example `src/wimax/examples/wimax-multicast.cc` shows how to specify a non-default channel or propagation model:

```cpp
class = CreateObject<SimpleOfdmWimaxChannel> ();
channel->SetPropagationModel (SimpleOfdmWimaxChannel::COST231_PROPAGATION);
ssDevs = wimax.Install (ssNodes,
    WimaxHelper::DEVICE_TYPE_SUBSCRIBER_STATION,
    WimaxHelper::SIMPLE_PHY_TYPE_OFDM,
    channel,
    scheduler);
```

Mobility is also supported in the same way as in Wifi models; see the `src/wimax/examples/wimax-multicast.cc`. Another important concept in WiMAX is that of a service flow. This is a unidirectional flow of packets with a set of QoS parameters such as traffic priority, rate, scheduling type, etc. The base station is responsible for issuing service flow identifiers and mapping them to WiMAX connections. The following code from `src/wimax/examples/wimax-multicast.cc` shows how this is configured from a helper level:

```cpp
ServiceFlow MulticastServiceFlow = wimax.CreateServiceFlow (ServiceFlow::SF_DIRECTION_DOWN,
    ServiceFlow::SF_TYPE_UGS,
    MulticastClassifier);
bs->GetServiceFlowManager ()->AddMulticastServiceFlow (MulticastServiceFlow, WimaxPhy::MODULATION_TYPE_QPSK_12);
```

### 25.3 Wimax Attributes

The WimaxNetDevice makes heavy use of the `ns-3 Attributes` subsystem for configuration and default value management. Presently, approximately 60 values are stored in this system.

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

- **NoiseFigure**: Loss (dB) in the Signal-to-Noise-Ratio due to non-idealities in the receiver.
- **TxPower**: Transmission power (dB)
- **G**: The ratio of CP time to useful time
- **txGain**: Transmission gain (dB)
- **RxGain**: Reception gain (dB)
- **Nfft**: FFT size
- **TraceFilePath**: Path to the directory containing SNR to block error rate files
For a full list of attributes in these models, consult the Doxygen page that lists all attributes for ns-3.

### 25.4 Wimax Tracing

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

Many ns-3 users use the built-in Pcap or Ascii tracing, and the WimaxHelper has similar APIs:

```cpp
AsciiTraceHelper ascii;
WimaxHelper wimax;
wimax.EnablePcap ("wimax-program", false);
wimax.EnableAsciiAll (ascii.CreateFileStream ("wimax-program.tr");
```

Unlike other helpers, there is also a special `EnableAsciiForConnection()` method that limits the ascii tracing to a specific device and connection.

These helpers access the low level trace sources that exist in the WiMAX physical layer, net device, and queue models. Like other ns-3 trace sources, users may hook their own functions to these trace sources if they want to do customized things based on the packet events. See the Doxygen List of trace sources for a complete list of these sources.

### 25.5 Wimax MAC model

The 802.16 model provided in ns-3 attempts to provide an accurate MAC and PHY level implementation of the 802.16 specification with the Point-to-Multipoint (PMP) mode and the WirelessMAN-OFDM PHY layer. The model is mainly composed of three layers:

- The convergence sublayer (CS)
- The MAC CP Common Part Sublayer (MAC-CPS)
- Physical (PHY) layer

The following figure `wimax-architecture` shows the relationships of these models.

#### 25.5.1 Convergence Sublayer

The Convergence sublayer (CS) provided with this module implements the Packet CS, designed to work with the packet-based protocols at higher layers. The CS is responsible of receiving packet from the higher layer and from peer stations, classifying packets to appropriate connections (or service flows) and processing packets. It keeps a mapping of transport connections to service flows. This enables the MAC CPS identifying the Quality of Service (QoS) parameters associated to a transport connection and ensuring the QoS requirements. The CS currently employs an IP classifier.

#### 25.5.2 IP Packet Classifier

An IP packet classifier is used to map incoming packets to appropriate connections based on a set of criteria. The classifier maintains a list of mapping rules which associate an IP flow (src IP address and mask, dst IP address and mask, src port range, dst port range and protocol) to one of the service flows. By analyzing the IP and the TCP/UDP headers the classifier will append the incoming packet (from the upper layer) to the queue of the appropriate WiMAX connection. Class `IpcsClassifier` and class `IpcsClassifierRecord` implement the classifier module for both SS and BS.
25.5. Wimax MAC model
25.5.3 MAC Common Part Sublayer

The MAC Common Part Sublayer (CPS) is the main sublayer of the IEEE 802.16 MAC and performs the fundamental functions of the MAC. The module implements the Point-Multi-Point (PMP) mode. In PMP mode BS is responsible of managing communication among multiple SSs. The key functionalities of the MAC CPS include framing and addressing, generation of MAC management messages, SS initialization and registration, service flow management, bandwidth management and scheduling services. Class WimaxNetDevice represents the MAC layer of a WiMAX network device. This class extends the class NetDevice of the ns-3 API that provides abstraction of a network device. Class WimaxNetDevice is further extended by class BaseStationNetDevice and class SubscriberStationNetDevice, defining MAC layers of BS and SS, respectively. Besides these main classes, the key functions of MAC are distributed to several other classes.

25.5.4 Framing and Management Messages

The module implements a frame as a fixed duration of time, i.e., frame boundaries are defined with respect to time. Each frame is further subdivided into downlink (DL) and uplink (UL) subframes. The module implements the Time Division Duplex (TDD) mode where DL and UL operate on same frequency but are separated in time. A number of DL and UL bursts are then allocated in DL and UL subframes, respectively. Since the standard allows sending and receiving bursts of packets in a given DL or UL burst, the unit of transmission at the MAC layer is a packet burst. The module implements a special PacketBurst data structure for this purpose. A packet burst is essentially a list of packets. The BS downlink and uplink schedulers, implemented by class BSScheduler and class UplinkScheduler, are responsible of generating DL and UL subframes, respectively. In the case of DL, the subframe is simulated by transmitting consecutive bursts (instances PacketBurst). In case of UL, the subframe is divided, with respect to time, into a number of slots. The bursts transmitted by the SSs in these slots are then aligned to slot boundaries. The frame is divided into integer number of symbols and Physical Slots (PS) which helps in managing bandwidth more effectively. The number of symbols per frame depends on the underlying implementation of the PHY layer. The size of a DL or UL burst is specified in units of symbols.

25.5.5 Network Entry and Initialization

The network entry and initialization phase is basically divided into two sub-phases, (1) scanning and synchronization and (2) initial ranging. The entire phase is performed by the LinkManager component of SS and BS. Once an SS wants to join the network, it first scans the downlink frequencies to search for a suitable channel. The search is complete as soon as it detects a PHY frame. The next step is to establish synchronization with the BS. Once SS receives a Downlink-MAP (DL-MAP) message the synchronization phase is complete and it remains synchronized as long as it keeps receiving DL-MAP and Downlink Channel Descriptor (DCD) messages. After the synchronization is established, SS waits for a Uplink Channel Descriptor (UCD) message to acquire uplink channel parameters. Once acquired, the first sub-phase of the network entry and initialization is complete. Once synchronization is achieved, the SS waits for a UL-MAP message to locate a special grant, called initial ranging interval, in the UL subframe. This grant is allocated by the BS Uplink Scheduler at regular intervals. Currently this interval is set to 0.5 ms, however the user is enabled to modify its value from the simulation script.

25.5.6 Connections and Addressing

All communication at the MAC layer is carried in terms of connections. The standard defines a connection as a unidirectional mapping between the SS and BS’s MAC entities for the transmission of traffic. The standard defines two types of connections: management connections for transmitting control messages and transport connections for data transmission. A connection is identified by a 16-bit Connection Identifier (CID). Class WimaxConnection and class Cid implement the connection and CID, respectively. Note that each connection maintains its own transmission queue where packets to transmit on that connection are queued. The ConnectionManager component of BS is responsible of creating and managing connections for all SSs.
The two key management connections defined by the standard, namely the Basic and Primary management connections, are created and allocated to the SS during the ranging process. Basic connection plays an important role throughout the operation of SS also because all (unidirectional) DL and UL grants are directed towards SS’s Basic CID. In addition to management connections, an SS may have one or more transport connections to send data packets. The Connection Manager component of SS manages the connections associated to SS. As defined by the standard, a management connection is bidirectional, i.e., a pair of downlink and uplink connections is represented by the same CID. This feature is implemented in a way that one connection (in DL direction) is created by the BS and upon receiving the CID the SS then creates an identical connection (in UL direction) with the same CID.

### 25.5.7 Scheduling Services

The module supports the four scheduling services defined by the 802.16-2004 standard:

- Unsolicited Grant Service (UGS)
- Real-Time Polling Services (rtPS)
- Non Real-Time Polling Services (nrtPS)
- Best Effort (BE)

These scheduling services behave differently with respect to how they request bandwidth as well as how the it is granted. Each service flow is associated to exactly one scheduling service, and the QoS parameter set associated to a service flow actually defines the scheduling service it belongs to. When a service flow is created the UplinkScheduler calculates necessary parameters such as grant size and grant interval based on QoS parameters associated to it.

### 25.5.8 WiMAX Uplink Scheduler Model

Uplink Scheduler at the BS decides which of the SSs will be assigned uplink allocations based on the QoS parameters associated to a service flow (or scheduling service) and bandwidth requests from the SSs. Uplink scheduler together with Bandwidth Manager implements the complete scheduling service functionality. The standard defines up to four scheduling services (BE, UGS, rtPS, nrtPS) for applications with different types of QoS requirements. The service flows of these scheduling services behave differently with respect to how they request for bandwidth as well as how the bandwidth is granted. The module supports all four scheduling services. Each service flow is associated to exactly one transport connection and one scheduling service. The QoS parameters associated to a service flow actually define the scheduling service it belongs to. Standard QoS parameters for UGS, rtPS, nrtPS and BE services, as specified in Tables 111a to 111d of the 802.16e amendment, are supported. When a service flow is created the uplink scheduler calculates necessary parameters such as grant size and allocation interval based on QoS parameters associated to it. The current WiMAX module provides three different versions of schedulers.

- The first one is a simple priority-based First Come First Serve (FCFS). For the real-time services (UGS and rtPS) the BS then allocates grants/polls on regular basis based on the calculated interval. For the non real-time services (nrtPS and BE) only minimum reserved bandwidth is guaranteed if available after servicing real-time flows. Note that not all of these parameters are utilized by the uplink scheduler. Also note that currently only service flow with fixed-size packet size are supported, as currently set up in simulation scenario with OnOff application of fixed packet size. This scheduler is implemented by class BSSchedulerSimple and class UplinkSchedulerSimple.

- The second one is similar to first scheduler except by rtPS service flow. All rtPS Connections are able to transmit all packet in the queue according to the available bandwidth. The bandwidth saturation control has been implemented to redistribute the effective available bandwidth to all rtPS that have at least one packet to transmit. The remaining bandwidth is allocated to nrtPS and BE Connections. This scheduler is implemented by class BSSchedulerRtps and class UplinkSchedulerRtps.

- The third one is a Migration-based Quality of Service uplink scheduler. This uplink scheduler uses three queues, the low priority queue, the intermediate queue and the high priority queue. The scheduler serves the requests
in strict priority order from the high priority queue to the low priority queue. The low priority queue stores the bandwidth requests of the BE service flow. The intermediate queue holds bandwidth requests sent by rtPS and by nrtPS connections. rtPS and nrtPS requests can migrate to the high priority queue to guarantee that their QoS requirements are met. Besides the requests migrated from the intermediate queue, the high priority queue stores periodic grants and unicast request opportunities that must be scheduled in the following frame. To guarantee the maximum delay requirement, the BS assigns a deadline to each rtPS bandwidth request in the intermediate queue. The minimum bandwidth requirement of both rtPS and nrtPS connections is guaranteed over a window of duration T. This scheduler is implemented by class UplinkSchedulerMBQoS.

25.5.9 WiMAX Outbound Schedulers Model

Besides the uplink scheduler these are the outbound schedulers at BS and SS side (BSScheduler and SSScheduler). The outbound schedulers decide which of the packets from the outbound queues will be transmitted in a given allocation. The outbound scheduler at the BS schedules the downlink traffic, i.e., packets to be transmitted to the SSs in the downlink subframe. Similarly the outbound scheduler at a SS schedules the packet to be transmitted in the uplink allocation assigned to that SS in the uplink subframe. All three schedulers have been implemented to work as FCFS scheduler, as they allocate grants starting from highest priority scheduling service to the lower priority one (UGS> rtPS> nrtPS> BE). The standard does not suggest any scheduling algorithm and instead leaves this decision up to the manufacturers. Of course more sophisticated algorithms can be added later if required.

25.6 WimaxChannel and WimaxPhy models

The module implements the Wireless MAN OFDM PHY specifications as the more relevant for implementation as it is the schema chosen by the WiMAX Forum. This specification is designed for non-light-of-sight (NLOS) including fixed and mobile broadband wireless access. The proposed model uses a 256 FFT processor, with 192 data subcarriers. It supports all the seven modulation and coding schemes specified by Wireless MAN-OFDM. It is composed of two parts: the channel model and the physical model.

25.7 Channel model

The channel model we propose is implemented by the class SimpleOFDMWimaxChannel which extends the class wimaxchannel. The channel entity has a private structure named m_phyList which handles all the physical devices connected to it. When a physical device sends a packet (FEC Block) to the channel, the channel handles the packet, and then for each physical device connected to it, it calculates the propagation delay, the path loss according to a given propagation model and eventually forwards the packet to the receiver device. The channel class uses the method GetDistanceFrom() to calculate the distance between two physical entities according to their 3D coordinates. The delay is computed as \( \text{delay} = \frac{\text{distance}}{C} \), where \( C \) is the speed of the light.

25.8 Physical model

The physical layer performs two main operations: (i) It receives a burst from a channel and forwards it to the MAC layer, (ii) it receives a burst from the MAC layer and transmits it on the channel. In order to reduce the simulation complexity of the WiMAX physical layer, we have chosen to model offline part of the physical layer. More specifically we have developed an OFDM simulator to generate trace files used by the reception process to evaluate if a FEC block can be correctly decoded or not.

Transmission Process: A burst is a set of WiMAX MAC PDUs. At the sending process, a burst is converted into bit-streams and then splitted into smaller FEC blocks which are then sent to the channel with a power equal P_tx.
Reception Process: The reception process includes the following operations:

1. Receive a FEC block from the channel.
2. Calculate the noise level.
3. Estimate the signal to noise ratio (SNR) with the following formula.
4. Determine if a FEC block can be correctly decoded.
5. Concatenate received FEC blocks to reconstruct the original burst.
6. Forward the burst to the upper layer.

The developed process to evaluate if a FEC block can be correctly received or not uses pre-generated traces. The trace files are generated by an external OFDM simulator (described later). A class named SNRToBlockErrorRateManager handles a repository containing seven trace files (one for each modulation and coding scheme). A repository is specific for a particular channel model.

A trace file is made of 6 columns. The first column provides the SNR value (1), whereas the other columns give respectively the bit error rate BER (2), the block error rate BlcER(3), the standard deviation on BlcER, and the confidence interval (4 and 5). These trace files are loaded into memory by the SNRToBlockErrorRateManager entity at the beginning of the simulation.

Currently, the first process uses the first and third columns to determine if a FEC block is correctly received. When the physical layer receives a packet with an SNR equal to SNR_rx, it asks the SNRToBlockErrorRateManager to return the corresponding block error rate BlcER. A random number RAND between 0 and 1 is then generated. If RAND is greater than BlcER, then the block is correctly received, otherwise the block is considered erroneous and is ignored.

The module provides defaults SNR to block error rate traces in default-traces.h. The traces have been generated by an External WiMAX OFDM simulator. The simulator is based on an external mathematics and signal processing library IT++ and includes: a random block generator, a Reed Solomon (RS) coder, a convolutional coder, an interleaver, a 256 FFT-based OFDM modulator, a multi-path channel simulator and an equalizer. The multipath channel is simulated using the TDL_channel class of the IT++ library.

Users can configure the module to use their own traces generated by another OFDM simulator or ideally by performing experiments in real environment. For this purpose, a path to a repository containing trace files should be provided. If no repository is provided the traces form default-traces.h will be loaded. A valid repository should contain 7 files, one for each modulation and coding scheme.

The names of the files should respect the following format: modulation0.txt for modulation 0, modulation1.txt for modulation 1 and so on... The file format should be as follows:

```
SNR_value1  BER  Blc_ER  STANDARD_DEVIATION  CONFIDENCE_INTERVAL1  CONFIDENCE_INTERVAL2
SNR_value2  BER  Blc_ER  STANDARD_DEVIATION  CONFIDENCE_INTERVAL1  CONFIDENCE_INTERVAL2
...           ...  ...           ...               ...               ...
```

25.8. Physical model
BIBLIOGRAPHY

[rfc3561] RFC 3561 *Ad hoc On-Demand Distance Vector (AODV) Routing*

[rfc3626] RFC 3626 *Optimized Link State Routing*

[ieee80211] IEEE Std 802.11-2007 *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*

[pei80211b] G. Pei and Tom Henderson, *Validation of ns-3 802.11b PHY model*

[pei80211ofdm] G. Pei and Tom Henderson, *Validation of OFDM error rate model in ns-3*
