# ns-3 Training

Session 2:  Monday May 11

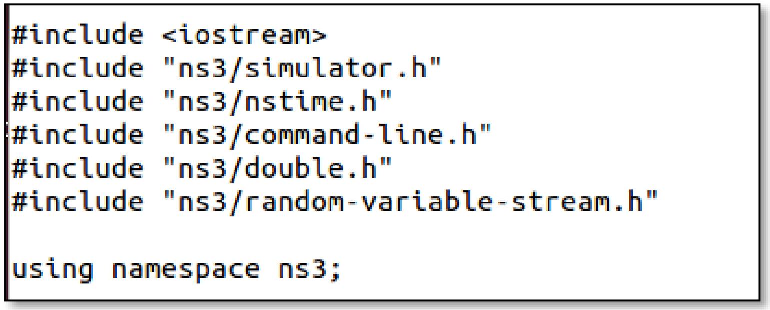
ns-3 Annual meeting
May 2015

# Simulator core

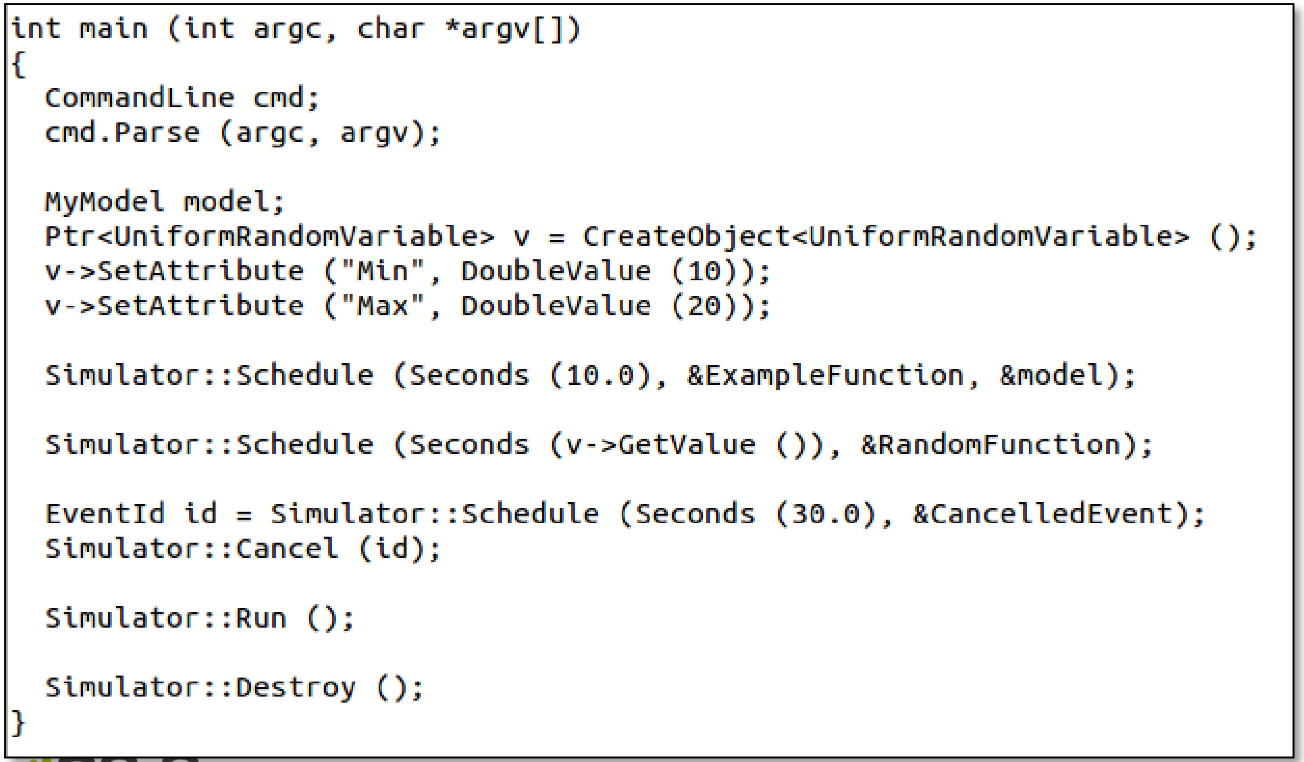
# Simulator core

- Simulation time
- Events
- Simulator and Scheduler
- Command line arguments
- Random variables

# Simulator example

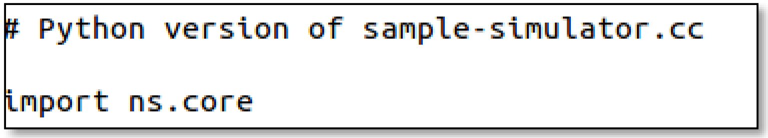


```
#include <iostream>
#include "ns3/simulator.h"
#include "ns3/nstime.h"
#include "ns3/command-line.h"
#include "ns3/double.h"
#include "ns3/random-variable-stream.h"

using namespace ns3;
```

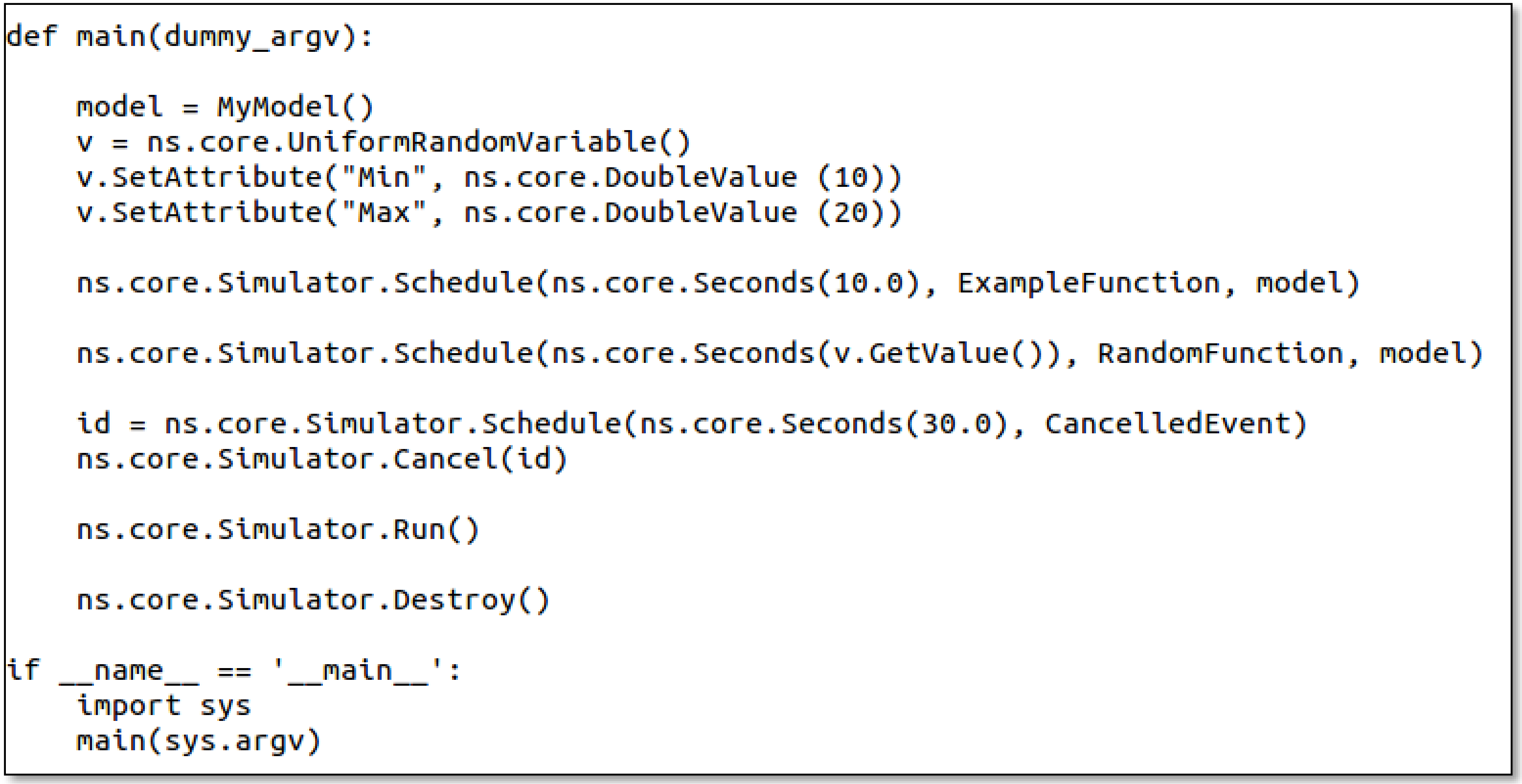


```
int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);

  MyModel model;
  Ptr<UniformRandomVariable> v = CreateObject<UniformRandomVariable> ();
  v->SetAttribute ("Min", DoubleValue (10));
  v->SetAttribute ("Max", DoubleValue (20));

  Simulator::Schedule (Seconds (10.0), &ExampleFunction, &model);

  Simulator::Schedule (Seconds (v->GetValue ()), &RandomFunction);

  EventId id = Simulator::Schedule (Seconds (30.0), &CancelledEvent);
  Simulator::Cancel (id);

  Simulator::Run ();

  Simulator::Destroy ();
}
```

# Simulator example (in Python)



```
# Python version of sample-simulator.cc

import ns.core
```



```
def main(dummy_argv):

    model = MyModel()
    v = ns.core.UniformRandomVariable()
    v.SetAttribute("Min", ns.core.DoubleValue (10))
    v.SetAttribute("Max", ns.core.DoubleValue (20))

    ns.core.Simulator.Schedule(ns.core.Seconds(10.0), ExampleFunction, model)

    ns.core.Simulator.Schedule(ns.core.Seconds(v.GetValue()), RandomFunction, model)

    id = ns.core.Simulator.Schedule(ns.core.Seconds(30.0), CancelledEvent)
    ns.core.Simulator.Cancel(id)

    ns.core.Simulator.Run()

    ns.core.Simulator.Destroy()

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

# Simulation program flow

Handle program inputs

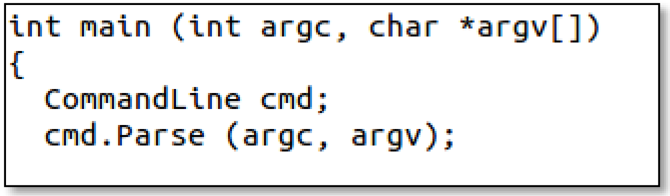
Configure topology

Run simulation

Process outputs

# Command-line arguments

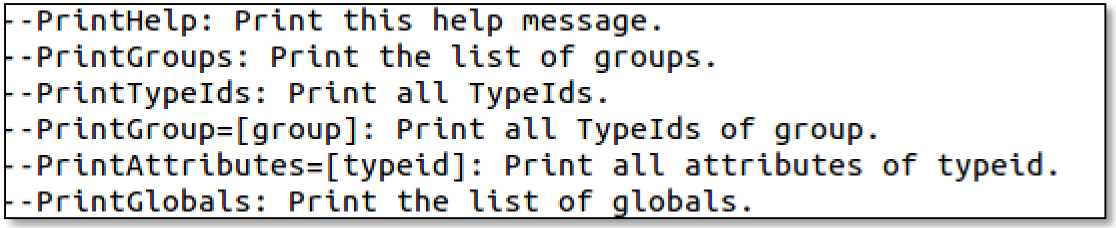
- Add CommandLine to your program if you want command-line argument parsing



```
int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);
```

- Passing --PrintHelp to programs will display command line options, if CommandLine is enabled

```
./waf --run "sample-simulator --PrintHelp"
```



```
--PrintHelp: Print this help message.
--PrintGroups: Print the list of groups.
--PrintTypeIds: Print all TypeIds.
--PrintGroup=[group]: Print all TypeIds of group.
--PrintAttributes=[typeid]: Print all attributes of typeid.
--PrintGlobals: Print the list of globals.
```

# Time in ns-3

- Time is stored as a large integer in ns-3
  - Minimize floating point discrepancies across platforms
- Special Time classes are provided to manipulate time (such as standard operators)
- Default time resolution is nanoseconds, but can be set to other resolutions
- Time objects can be set by floating-point values and can export floating-point values

```
double timeDouble = t.GetSeconds();
```

# Events in ns-3

- Events are just function calls that execute at a simulated time
  - i.e. callbacks
  - another difference compared to other simulators, which often use special "event handlers" in each model
- Events have IDs to allow them to be cancelled or to test their status

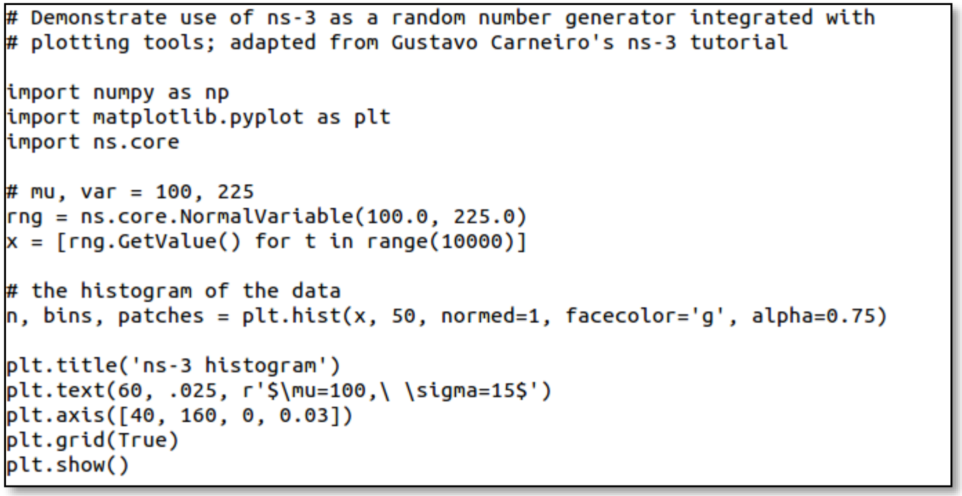
# Simulator and Schedulers

- The Simulator class holds a scheduler, and provides the API to schedule events, start, stop, and cleanup memory
- Several scheduler data structures (calendar, heap, list, map) are possible
- "RealTime" simulation implementation aligns the simulation time to wall-clock time
  - two policies (hard and soft limit) available when the simulation and real time diverge

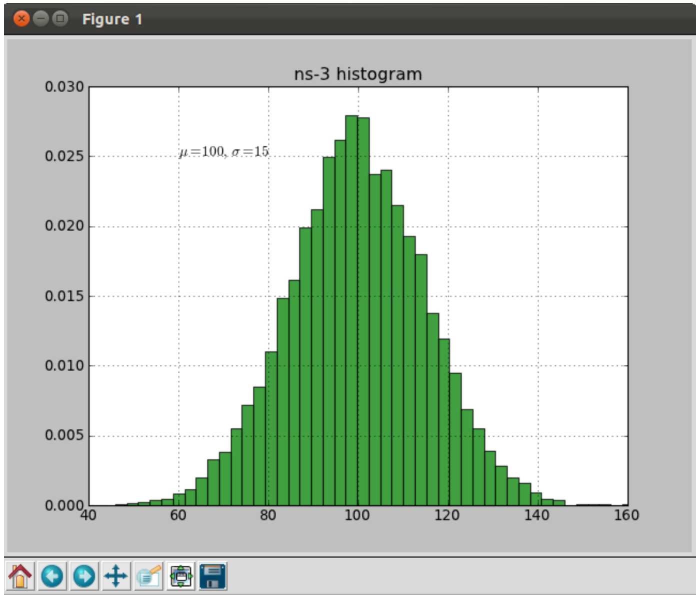
# Random Variables

from src/core/examples/sample-rng-plot.py

- Currently implemented distributions
  - Uniform: values uniformly distributed in an interval
  - Constant: value is always the same (not really random)
  - Sequential: return a sequential list of predefined values
  - Exponential: exponential distribution (poisson process)
  - Normal (gaussian), Log-Normal, Pareto, Weibull, triangular



```
# Demonstrate use of ns-3 as a random number generator integrated with
# plotting tools; adapted from Gustavo Carneiro's ns-3 tutorial

import numpy as np
import matplotlib.pyplot as plt
import ns.core

# mu, var = 100, 225
rng = ns.core.NormalVariable(100.0, 225.0)
x = [rng.GetValue() for t in range(10000)]

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.title('ns-3 histogram')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```



# Random variables and independent replications

- Many simulation uses involve running a number of *independent replications* of the same scenario

- In ns-3, this is typically performed by incrementing the simulation *run number*
  - *not by changing seeds*

# ns-3 random number generator

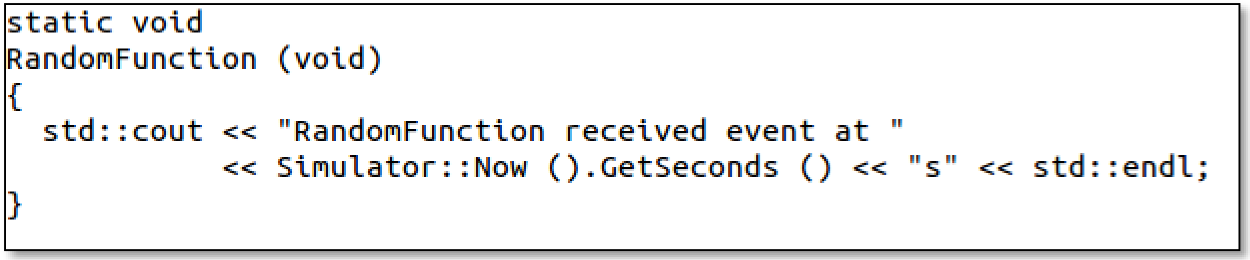
- Uses the MRG32k3a generator from Pierre L'Ecuyer
  - http://www.iro.umontreal.ca/~lecuyer/myftp/papers/streams00.pdf
  - Period of PRNG is 3.1x10^57
- Partitions a pseudo-random number generator into <u>uncorrelated</u> *streams* and *substreams*
  - Each RandomVariableStream gets its own stream
  - This stream partitioned into substreams

# Run number vs. seed

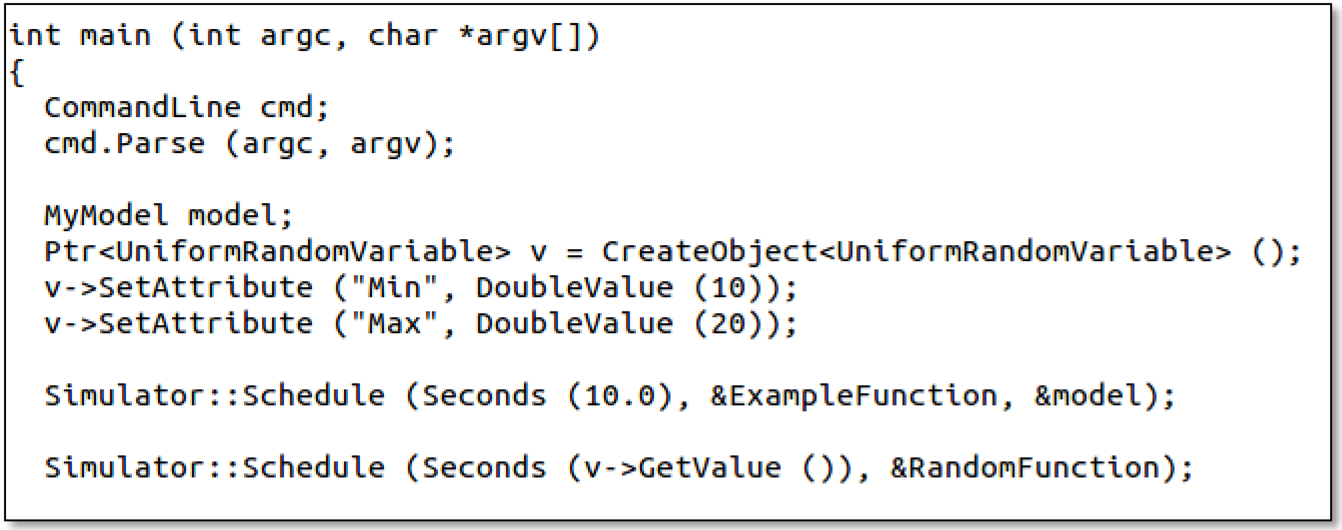
- If you increment the seed of the PRNG, the streams of random variable objects across different runs are not guaranteed to be uncorrelated
- If you fix the seed, but increment the run number, you will get an uncorrelated substream

# Putting it together

- Example of scheduled event



```
static void
RandomFunction (void)
{
  std::cout << "RandomFunction received event at "
            << Simulator::Now ().GetSeconds () << "s" << std::endl;
}
```



```
int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);

  MyModel model;
  Ptr<UniformRandomVariable> v = CreateObject<UniformRandomVariable> ();
  v->SetAttribute ("Min", DoubleValue (10));
  v->SetAttribute ("Max", DoubleValue (20));

  Simulator::Schedule (Seconds (10.0), &ExampleFunction, &model);

  Simulator::Schedule (Seconds (v->GetValue ()), &RandomFunction);
```

Demo real-time, command-line, random variables...

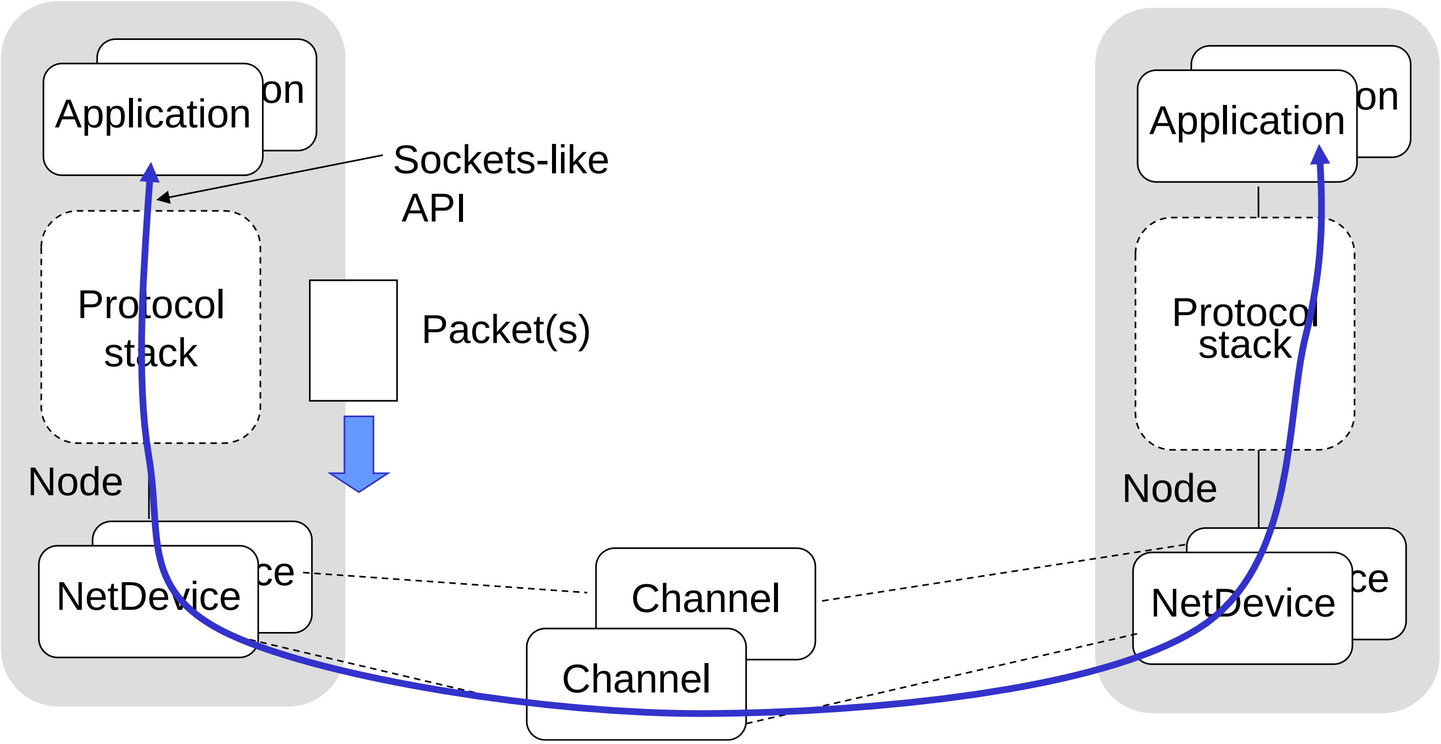
# Nodes and Devices

# Example walkthrough

- This section progressively builds up a simple ns-3 example, explaining concepts along the way
- Files for these programs are available on the ns-3 wiki

# Example program

- `wns3-version1.cc`
  - Link found on wiki page
  - Place program in scratch/ folder



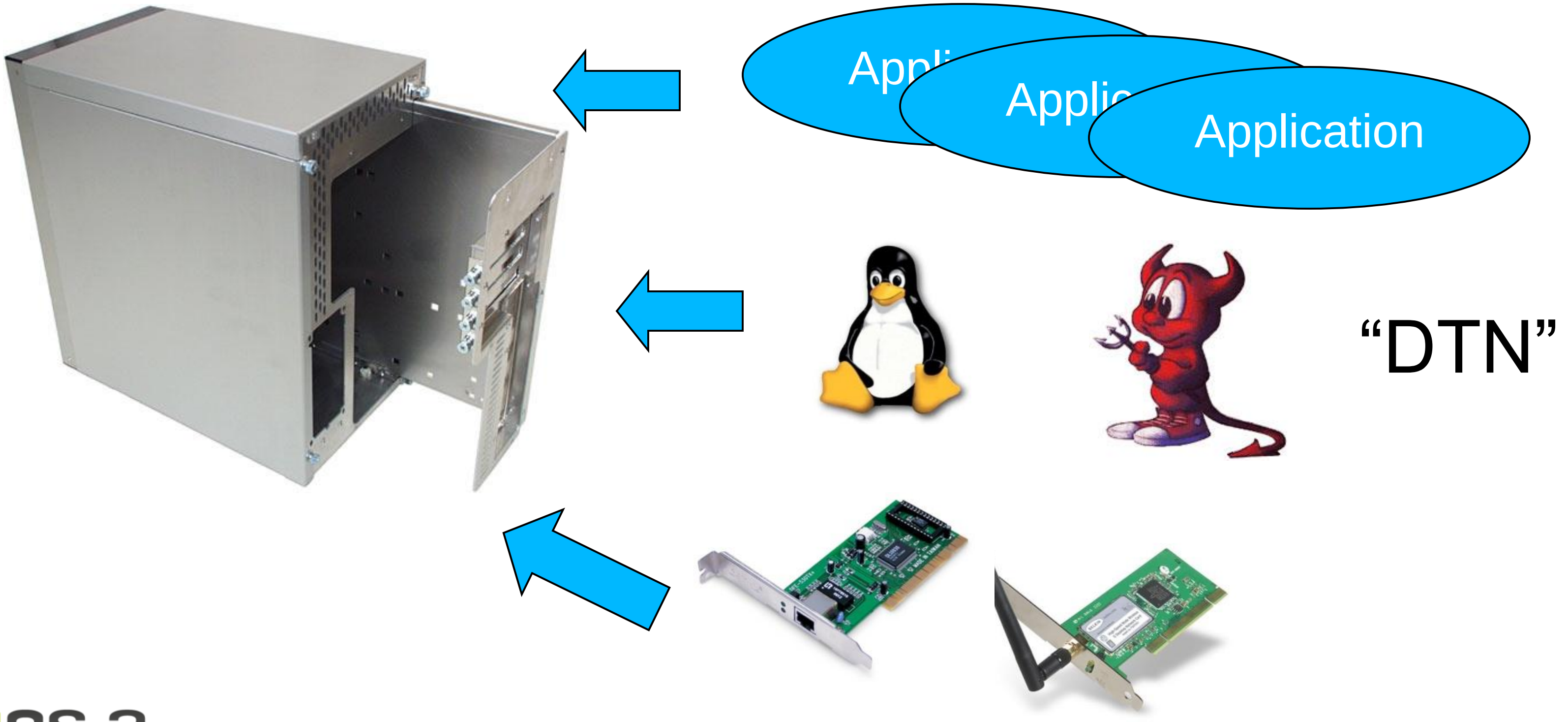
# Fundamentals

Key objects in the simulator are Nodes, Packets, and Channels

Nodes contain Applications, “stacks”, and NetDevices

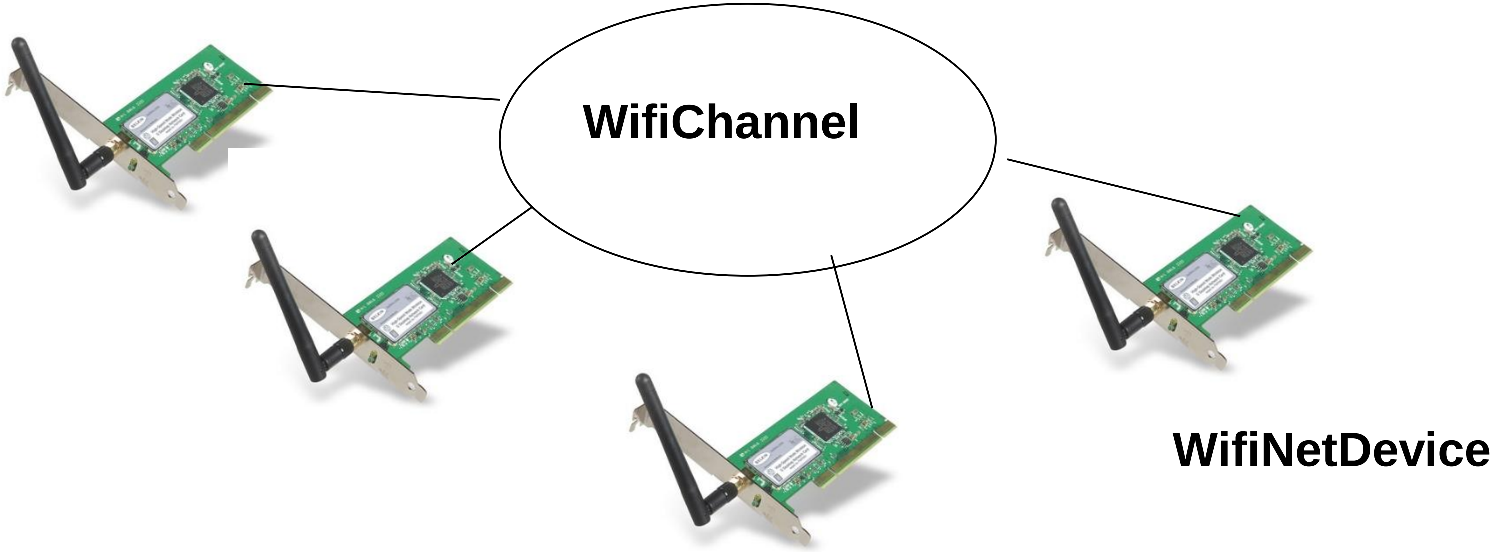
# Node basics

A Node is a shell of a computer to which applications, stacks, and NICs are added



# NetDevices and Channels

NetDevices are strongly bound to Channels of a matching type



- ns-3 Spectrum models relax this assumption

Nodes are architected for multiple interfaces

# Internet Stack

- Internet Stack
  - Provides IPv4 and some IPv6 models currently
- No non-IP stacks ns-3 until 802.15.4 was introduced in ns-3.20
  - but no dependency on IP in the devices, Node, Packet, etc.  (partly due to the object aggregation system)

# Other basic models in ns-3

- Devices
  - WiFi, WiMAX, CSMA, Point-to-point, Bridge
- Error models and queues
- Applications
  - echo servers, traffic generator
- Mobility models
- Packet routing
  - OLSR, AODV, DSR, DSDV, Static, Nix-Vector, Global (link state)

# Structure of an ns-3 program

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

  // Set default attribute values

  // Parse command-line arguments

  // Configure the topology; nodes, channels, devices, mobility

  // Add (Internet) stack to nodes

  // Configure IP addressing and routing

  // Add and configure applications

  // Configure tracing

  // Run simulation
}
```

# Helper API

- The ns-3 “helper API” provides a set of classes and methods that make common operations easier than using the low-level API
- Consists of:
  - container objects
  - helper classes
- The helper API is implemented using the low-level API
- Users are encouraged to contribute or propose improvements to the ns-3 helper API

# Containers

- Containers are part of the ns-3 “helper API”
- Containers group similar objects, for convenience
    - They are often implemented using C++ std containers
- Container objects also are intended to provide more basic (typical) API

# The Helper API (vs. low-level API)

- Is not generic
- Does not try to allow code reuse
- Provides simple 'syntactical sugar' to make simulation scripts look nicer and easier to read for network researchers
- Each function applies a single operation on a "set of same objects”
- A typical operation is "Install()"

# Helper Objects

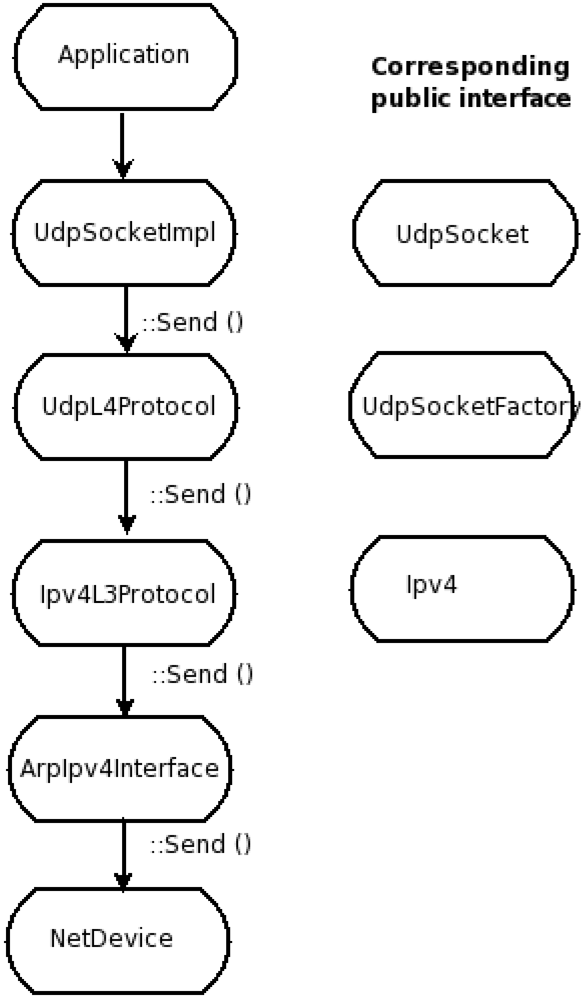
- NodeContainer: vector of Ptr<Node>
- NetDeviceContainer: vector of Ptr<NetDevice>
- InternetStackHelper
- WifiHelper
- MobilityHelper
- OlsrHelper
- ... Each model provides a helper class

# Installation onto containers

- Installing models into containers, and handling containers, is a key API theme

```
NodeContainer c;
c.Create (numNodes);
...
mobility.Install (c);
...
internet.Install (c);
...
```

# Internet stack



- The public interface of the Internet stack is defined (abstract base classes) in src/network/model directory
- The intent is to support multiple implementations
- The default ns-3 Internet stack is implemented in src/internet-stack

# Example program iterations

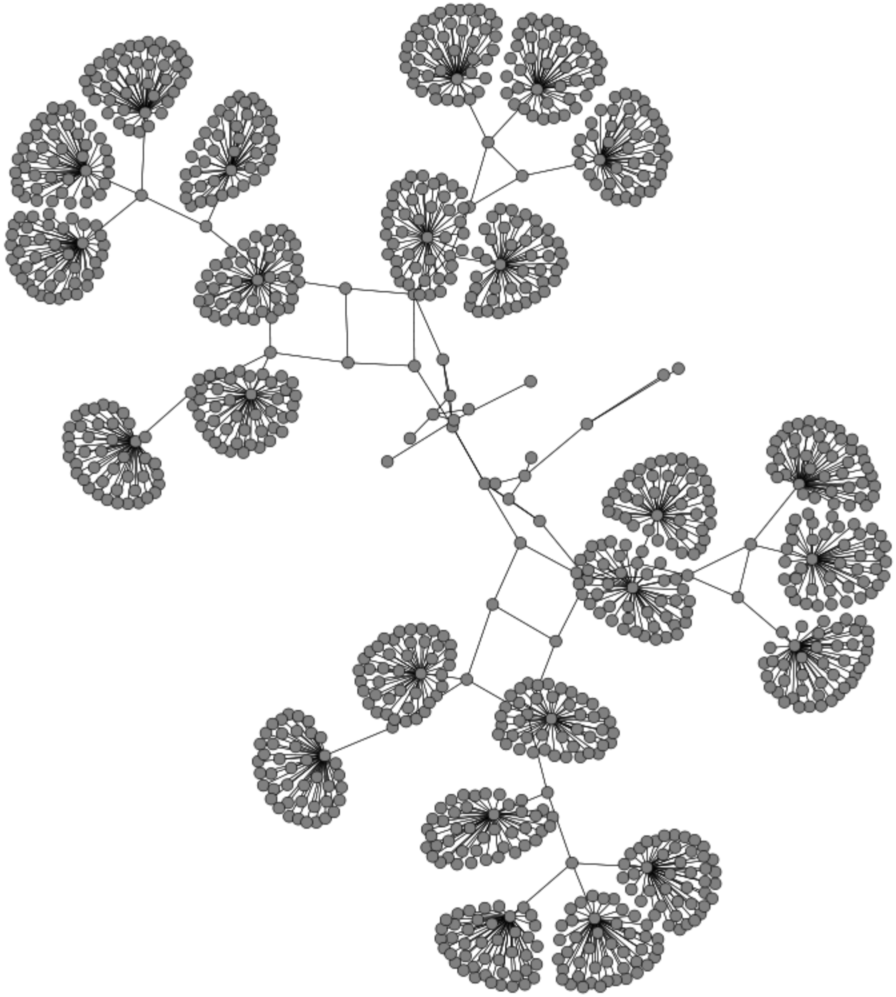
- Walk through four additional revisions of the example program
  - `wns3-version2.cc`
  - `wns3-version3.cc`
  - `wns3-version4.cc`

# Visualization

# PyViz overview

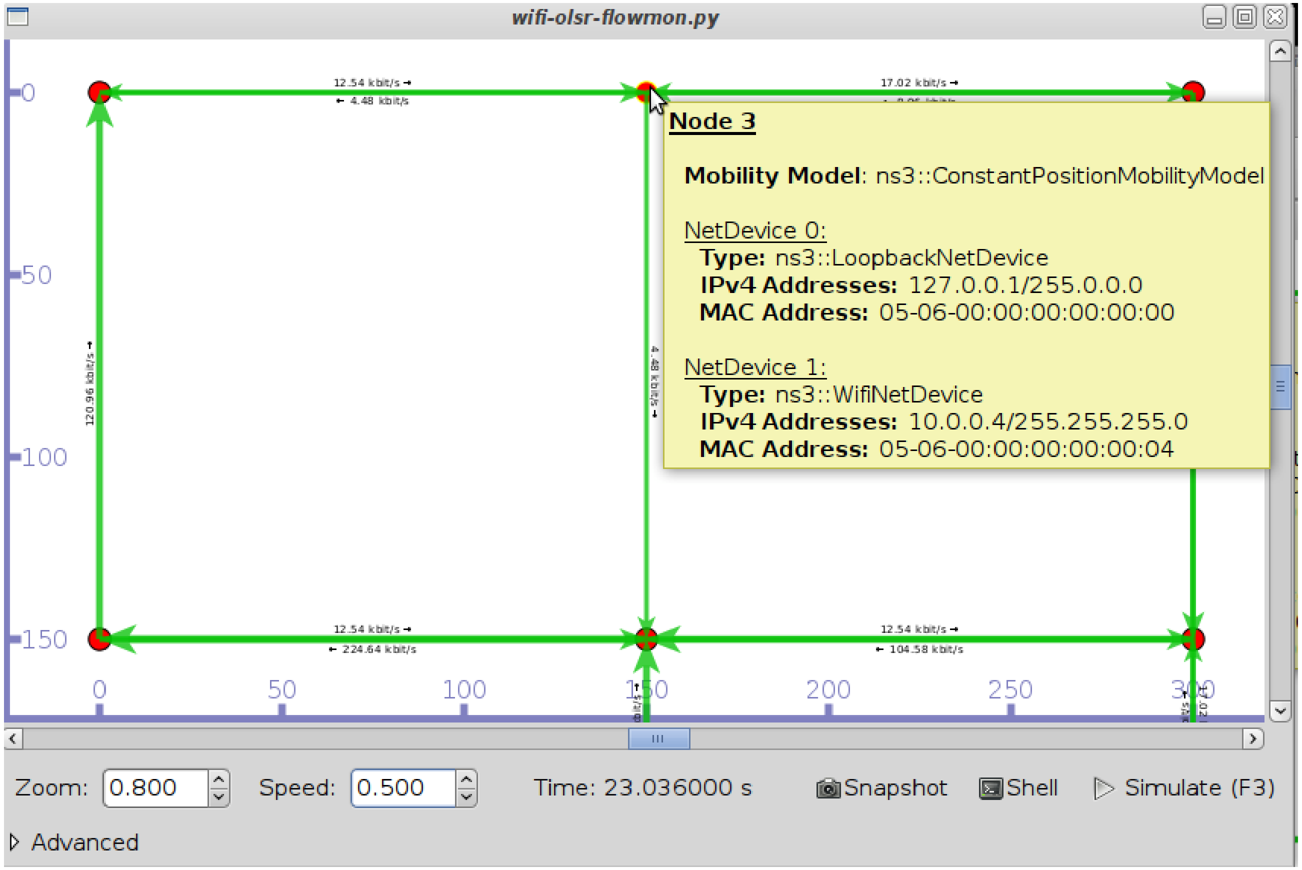
- Developed by Gustavo Carneiro
- Live simulation visualizer (no trace files)
- Useful for debugging
  - mobility model behavior
  - where are packets being dropped?
- Built-in interactive Python console to debug the state of running objects
- Works with Python and C++ programs

# Pyviz screenshot (Graphviz layout)



# Pyviz and FlowMonitor

- src/flow-monitor/examples/wifi-olsr-flowmon.py



**Node 3**

**Mobility Model**: ns3::ConstantPositionMobilityModel

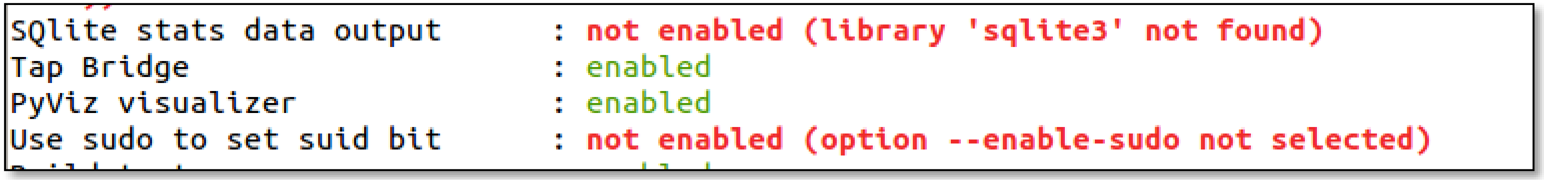
NetDevice 0:
**Type:** ns3::LoopbackNetDevice
**IPv4 Addresses:** 127.0.0.1/255.0.0.0
**MAC Address:** 05-06-00:00:00:00:00:00

NetDevice 1:
**Type:** ns3::WifiNetDevice
**IPv4 Addresses:** 10.0.0.4/255.255.255.0
**MAC Address:** 05-06-00:00:00:00:00:04

Zoom: 0.800 Speed: 0.500 Time: 23.036000 s

# Enabling PyViz in your simulations

- Make sure PyViz is enabled in the build



```
SQlite stats data output      : not enabled (library 'sqlite3' not found)
Tap Bridge                    : enabled
PyViz visualizer              : enabled
Use sudo to set suid bit      : not enabled (option --enable-sudo not selected)
```

- If program supports CommandLine parsing, pass the option

```
--SimulatorImplementationType=
ns3::VisualSimulatorImpl
```

- Alternatively, pass the "--vis" option

# FlowMonitor

- Network monitoring framework found in `src/flow-monitor/`
- Goals:
  - detect all flows passing through network
  - stores metrics for analysis such as bitrates, duration, delays, packet sizes, packet loss ratios

G. Carneiro, P. Fortuna, M. Ricardo, "FlowMonitor-- a network monitoring framework for the Network Simulator ns-3," Proceedings of NSTools 2009.

# FlowMonitor architecture

- Basic classes
  - – FlowMonitor
  - – FlowProbe
  - – FlowClassifier
  - – FlowMonitorHelper
- IPv6 coming in ns-3.20 release

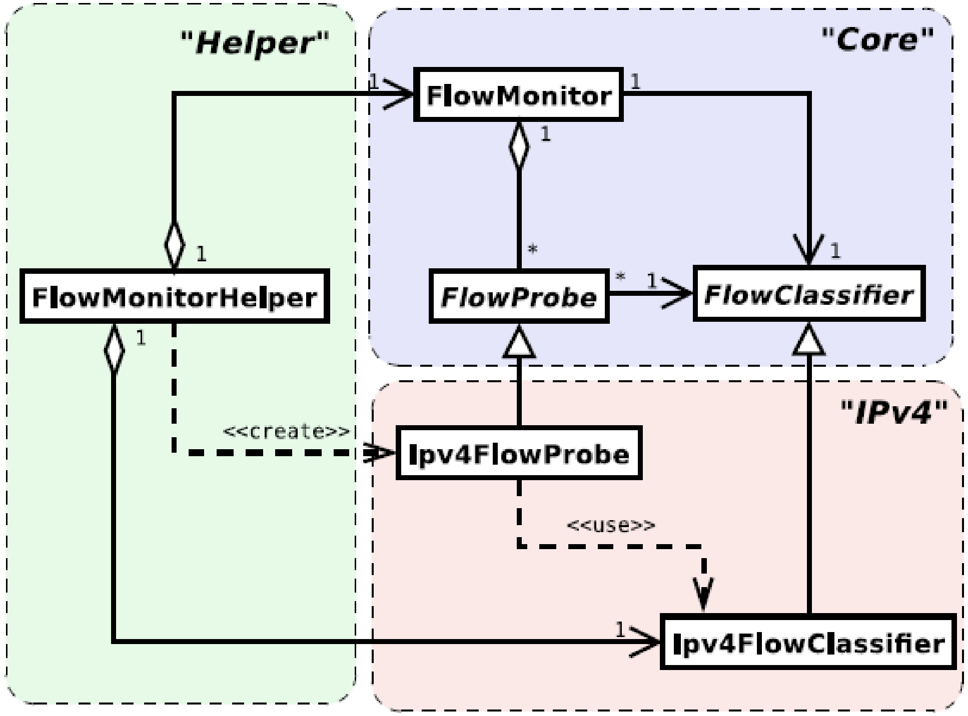
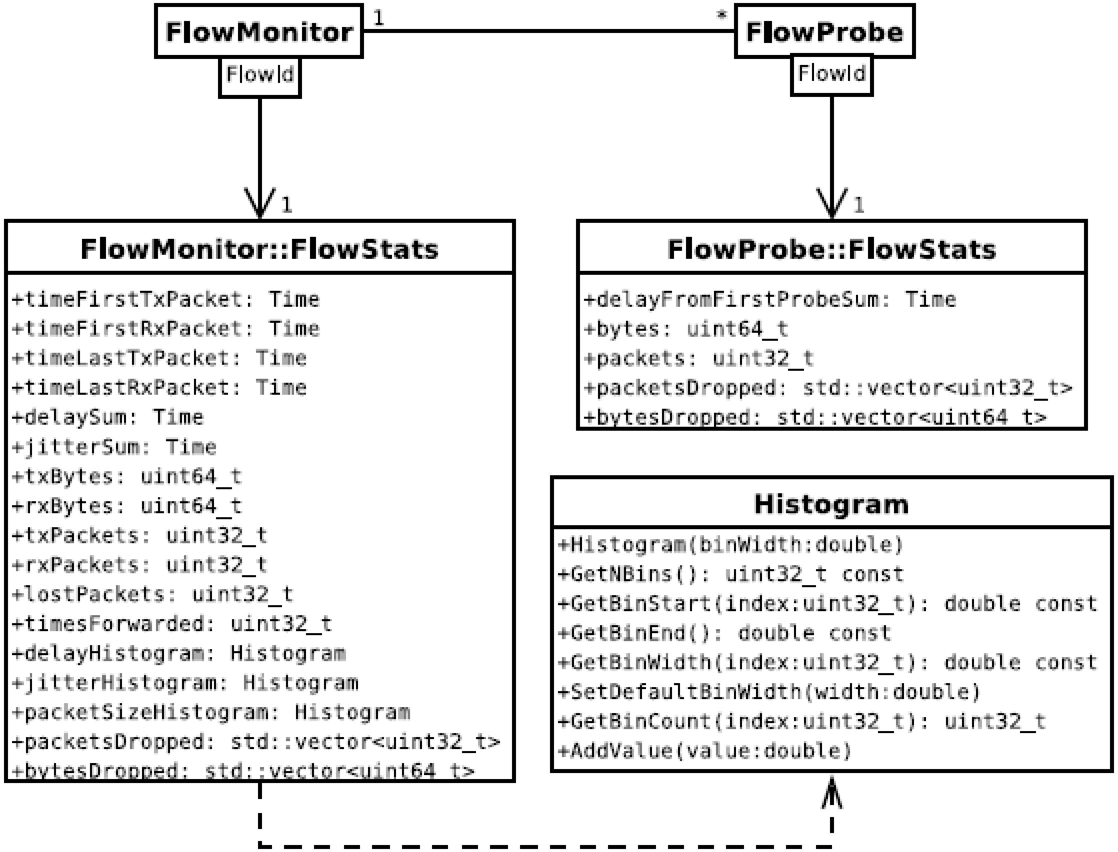


Figure credit:  G. Carneiro, P. Fortuna, M. Ricardo, "FlowMonitor-- a network monitoring framework for the Network Simulator ns-3," Proceedings of NSTools 2009.

# FlowMonitor statistics

- Statistics gathered



**FlowMonitor::FlowStats**

+timeFirstTxPacket: Time
+timeFirstRxPacket: Time
+timeLastTxPacket: Time
+timeLastRxPacket: Time
+delaySum: Time
+jitterSum: Time
+txBytes: uint64_t
+rxBytes: uint64_t
+txPackets: uint32_t
+rxPackets: uint32_t
+lostPackets: uint32_t
+timesForwarded: uint32_t
+delayHistogram: Histogram
+jitterHistogram: Histogram
+packetSizeHistogram: Histogram
+packetsDropped: std::vector<uint32_t>
+bytesDropped: std::vector<uint64_t>

**FlowProbe::FlowStats**

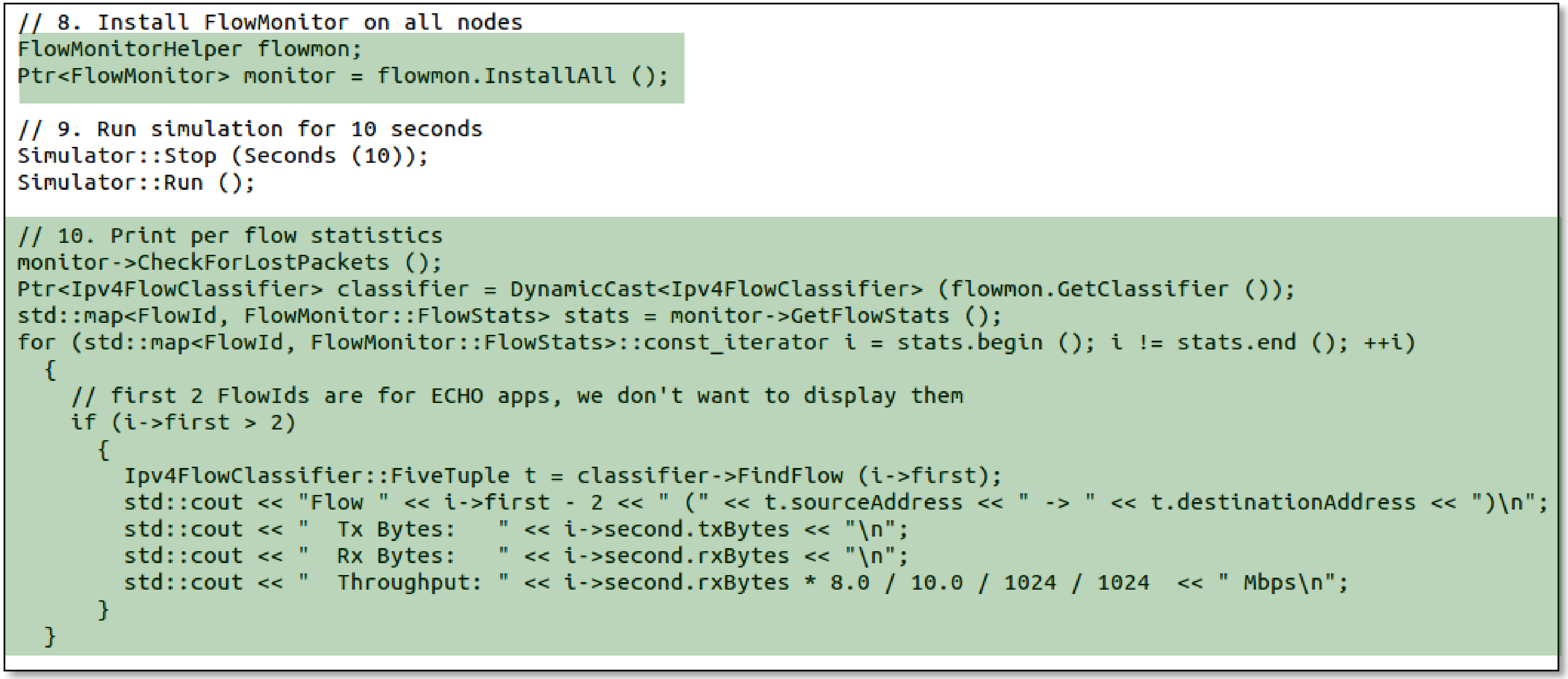
+delayFromFirstProbeSum: Time
+bytes: uint64_t
+packets: uint32_t
+packetsDropped: std::vector<uint32_t>
+bytesDropped: std::vector<uint64_t>

**Histogram**

+Histogram(binWidth:double)
+GetNBins(): uint32_t const
+GetBinStart(index:uint32_t): double const
+GetBinEnd(): double const
+GetBinWidth(index:uint32_t): double const
+SetDefaultBinWidth(width:double)
+GetBinCount(index:uint32_t): uint32_t
+AddValue(value:double)

# FlowMonitor configuration

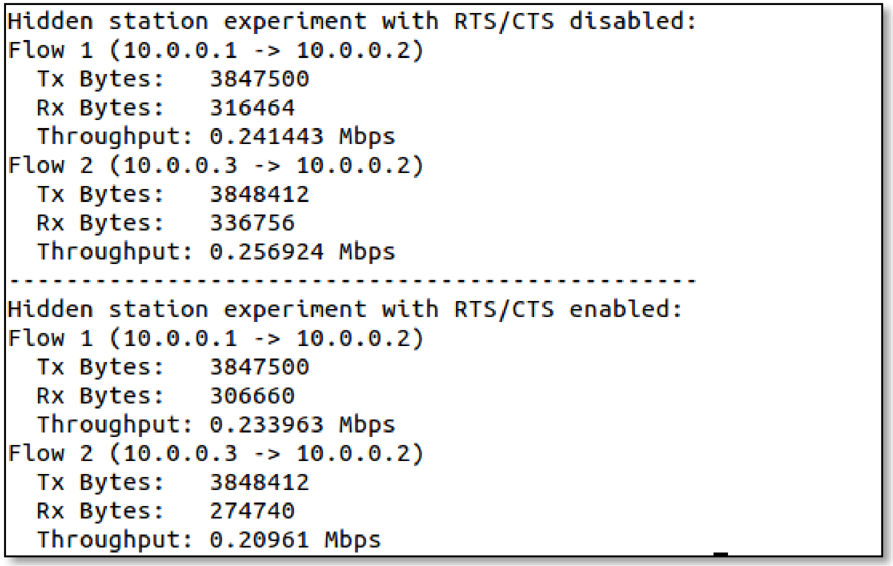
- `example/wireless/wifi-hidden-terminal.cc`



```
// 8. Install FlowMonitor on all nodes
FlowMonitorHelper flowmon;
Ptr<FlowMonitor> monitor = flowmon.InstallAll ();

// 9. Run simulation for 10 seconds
Simulator::Stop (Seconds (10));
Simulator::Run ();

// 10. Print per flow statistics
monitor->CheckForLostPackets ();
Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (flowmon.GetClassifier ());
std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();
for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin (); i != stats.end (); ++i)
  {
    // first 2 FlowIds are for ECHO apps, we don't want to display them
    if (i->first > 2)
      {
        Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);
        std::cout << "Flow " << i->first - 2 << " (" << t.sourceAddress << " -> " << t.destinationAddress << ")\n";
        std::cout << "  Tx Bytes:   " << i->second.txBytes << "\n";
        std::cout << "  Rx Bytes:   " << i->second.rxBytes << "\n";
        std::cout << "  Throughput: " << i->second.rxBytes * 8.0 / 10.0 / 1024 / 1024  << " Mbps\n";
      }
  }
```

# FlowMonitor output

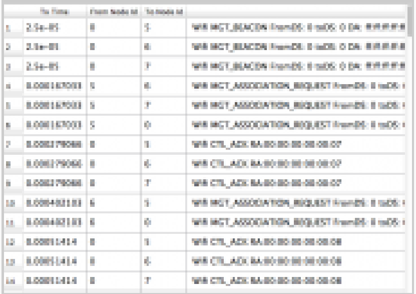
- This program exports statistics to stdout
- Other examples integrate with PyViz

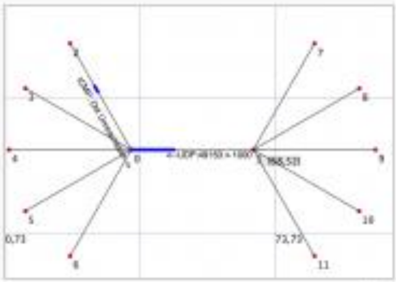


```
Hidden station experiment with RTS/CTS disabled:
Flow 1 (10.0.0.1 -> 10.0.0.2)
  Tx Bytes:   3847500
  Rx Bytes:   316464
  Throughput: 0.241443 Mbps
Flow 2 (10.0.0.3 -> 10.0.0.2)
  Tx Bytes:   3848412
  Rx Bytes:   336756
  Throughput: 0.256924 Mbps
------------------------------------------------
Hidden station experiment with RTS/CTS enabled:
Flow 1 (10.0.0.1 -> 10.0.0.2)
  Tx Bytes:   3847500
  Rx Bytes:   306660
  Throughput: 0.233963 Mbps
Flow 2 (10.0.0.3 -> 10.0.0.2)
  Tx Bytes:   3848412
  Rx Bytes:   274740
  Throughput: 0.20961 Mbps
```

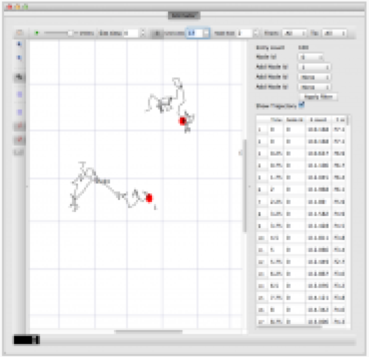
# NetAnim

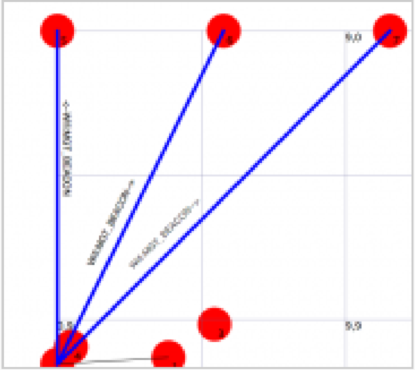
- "NetAnim" by George Riley and John Abraham


Packet Statistics

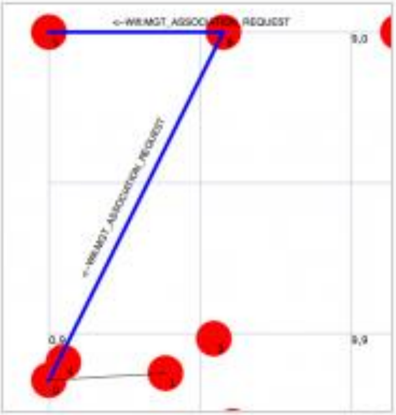


Dumbbell

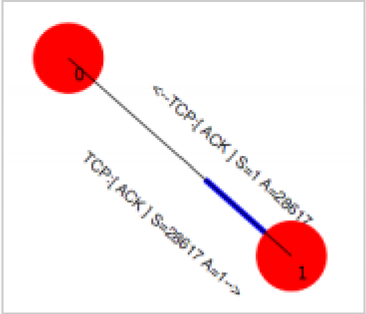

Node Trajectory



Wifi Beacon



Wifi Assoc



TCP flags

# NetAnim key features

- Animate packets over wired-links and wireless-links
  - limited support for LTE traces
- Packet timeline with regex filter on packet meta-data.
- Node position statistics with node trajectory plotting (path of a mobile node).
- Print brief packet-meta data on packets

# Placeholder for netanim videos

# ns-3 Objects

# Object metadata system

- ns-3 is, at heart, a C++ object system
- ns-3 objects that inherit from base class ns3::Object get several additional features
  - smart-pointer memory management (Class Ptr)
  - dynamic run-time object aggregation
  - an attribute system

# Smart pointers

- Smart pointers in ns-3 use reference counting to improve memory management
- The class `ns3::Ptr` is semantically similar to a traditional pointer, but the object pointed to will be deleted when all references to the pointer are gone
- ns-3 heap-allocated objects should use the templated Create<>() or CreateObject<> () methods

# Examples

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

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

(instead of `Packet* = new Packet;`)

why Create<> vs CreateObject<>?

- two different base classes; generally use CreateObject<>(), but Create<> for Packet

# Dynamic run-time object aggregation

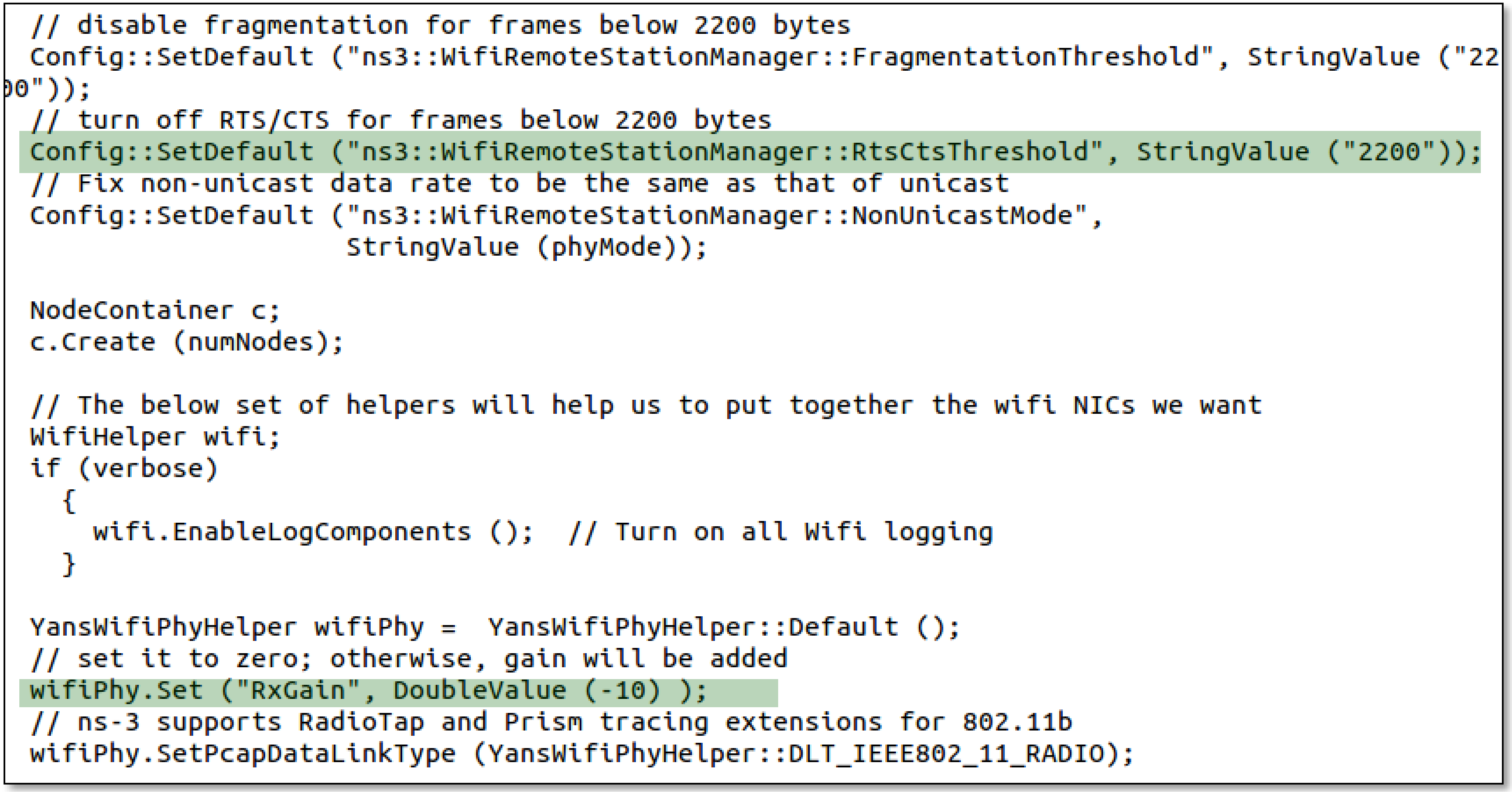
- This feature is similar to "Component Object Model (COM)"-- allows interfaces (objects) to be aggregated at run-time instead of at compile time
- Useful for binding dissimilar objects together without adding pointers to each other in the classes

# Usage

- ns-3 Node protocol stacks are added via aggregation
  - The IP stack can be found from a Node pointer without class Node knowing about it
- Energy models are typically aggregated to nodes
- To find interfaces, use GetObject<>(); e.g.

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

# Attributes and default values



```
// disable fragmentation for frames below 2200 bytes
Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold", StringValue ("22
00"));
// turn off RTS/CTS for frames below 2200 bytes
Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold", StringValue ("2200"));
// Fix non-unicast data rate to be the same as that of unicast
Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode",
                    StringValue (phyMode));

NodeContainer c;
c.Create (numNodes);

// The below set of helpers will help us to put together the wifi NICs we want
WifiHelper wifi;
if (verbose)
  {
    wifi.EnableLogComponents ();  // Turn on all Wifi logging
  }

YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
// set it to zero; otherwise, gain will be added
wifiPhy.Set ("RxGain", DoubleValue (-10) );
// ns-3 supports RadioTap and Prism tracing extensions for 802.11b
wifiPhy.SetPcapDataLinkType (YansWifiPhyHelper::DLT_IEEE802_11_RADIO);
```

# ns-3 attribute system

Problem: Researchers want to identify all of the values affecting the results of their simulations

- and configure them easily

ns-3 solution: Each ns-3 object has a set of attributes:

- A name, help text
- A type
- An initial value

- Control all simulation parameters for static objects
- Dump and read them all in configuration files
- Visualize them in a GUI
- Makes it easy to verify the parameters of a simulation

# Short digression: Object metadata system

- ns-3 is, at heart, a C++ object system
- ns-3 objects that inherit from base class ns3::Object get several additional features
  - dynamic run-time object aggregation
  - an attribute system ⬅
  - smart-pointer memory management (Class Ptr)

We focus here on the attribute system

# Use cases for attributes

- An Attribute represents a value in our system
- An Attribute can be connected to an underlying variable or function
  - e.g. TcpSocket::m_cwnd;
  - or a trace source

# Use cases for attributes (cont.)

- What would users like to do?
  - Know what are all the attributes that affect the simulation at run time
  - Set a default initial value for a variable
  - Set or get the current value of a variable
  - Initialize the value of a variable when a constructor is called
- The attribute system is a unified way of handling these functions

# How to handle attributes

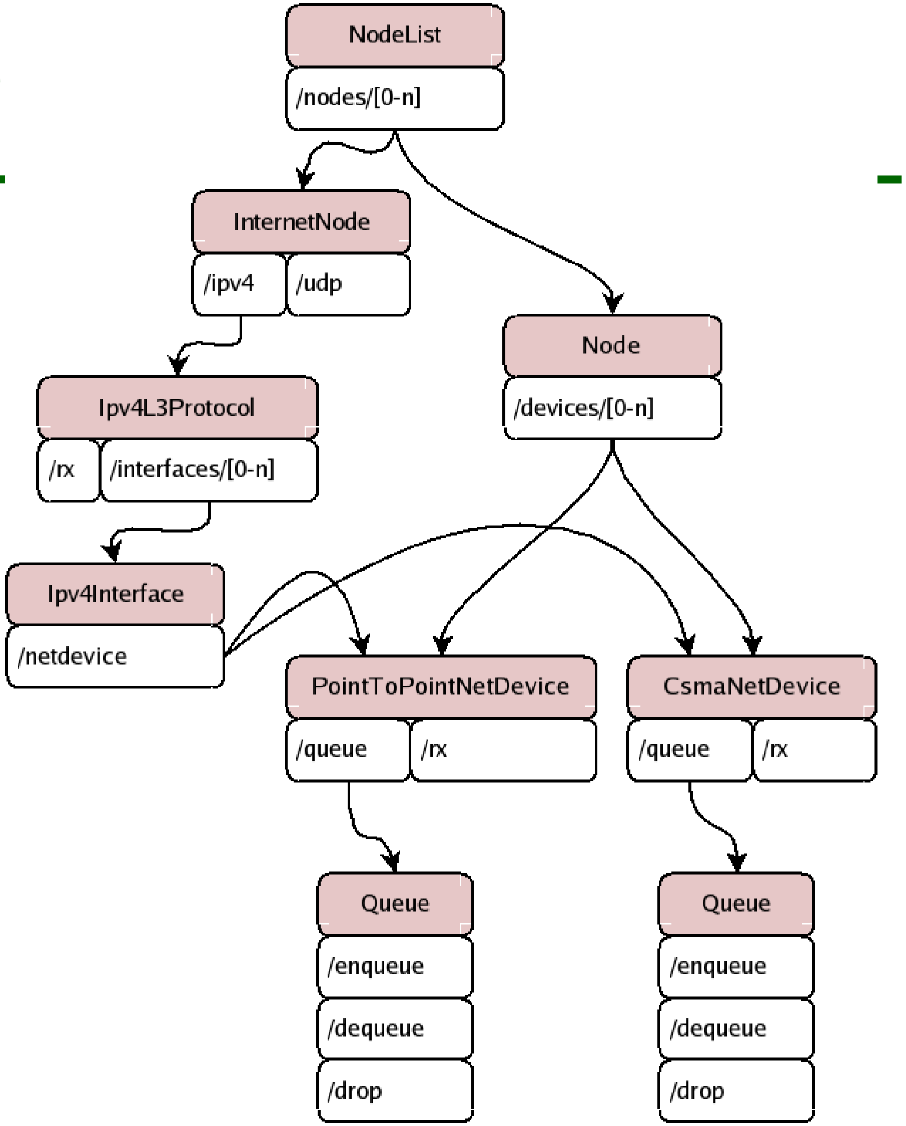
- The traditional C++ way:
  - export attributes as part of a class's public API
  - walk pointer chains (and iterators, when needed) to find what you need
  - use static variables for defaults
- The attribute system provides a more convenient API to the user to do these things

# Navigating the attributes

- Attributes are exported into a string-based namespace, with filesystem-like paths
  - namespace supports regular expressions
- Attributes also can be used without the paths
  - e.g. `“ns3::WifiPhy::TxGain”`
- A Config class allows users to manipulate the attributes

# Attribute namespace

- strings are used to describe paths through the namespace



`Config::Set ("/NodeList/1/$ns3::Ns3NscStack<linux2.6.26>/net.ipv4.tcp_sack", StringValue ("0"));`

# Navigating the attributes using paths

- Examples:
  - Nodes with NodeIds 1, 3, 4, 5, 8, 9, 10, 11:

    `“/NodeList/[3-5]|[8-11]|1”`
  - UdpL4Protocol object instance aggregated to matching nodes:

    `“/$ns3::UdpL4Protocol”`

# What users will do

- e.g.: Set a default initial value for a variable

```
Config::Set (“ns3::YansWifiPhy::TxGain”,
  DoubleValue (1.0));
```

- Syntax also supports string values:

```
Config::Set (“YansWifiPhy::TxGain”,
  StringValue (“1.0”));
```

Attribute

Value

# Fine-grained attribute handling

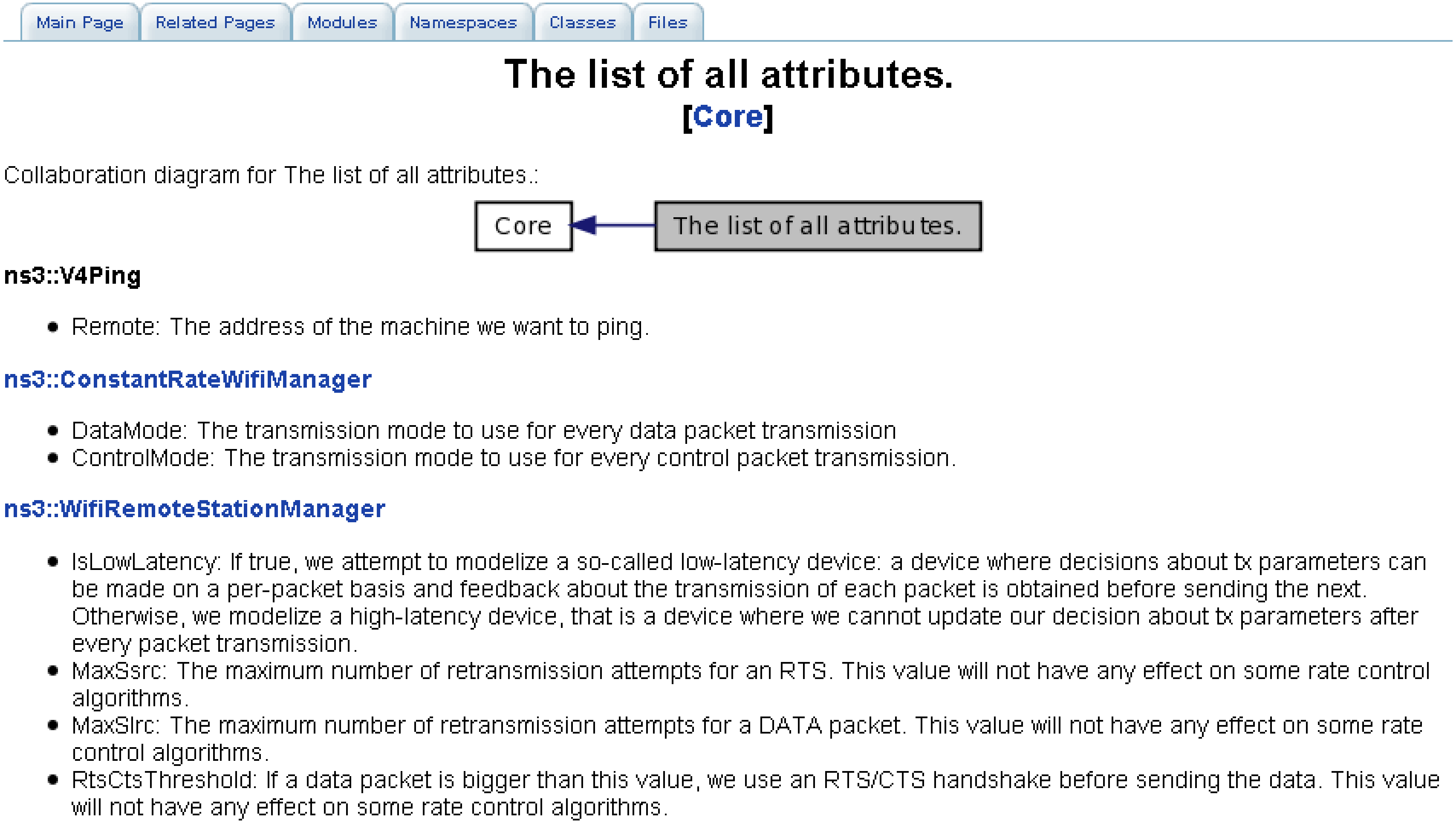
- Set or get the current value of a variable
  - Here, one needs the path in the namespace to the right instance of the object

```
Config::SetAttribute(“/NodeList/5/DeviceList/3/$n
  s3::WifiNetDevice/Phy/$ns3::YansWifiPhy/TxGain”,
  DoubleValue(1.0));
```

```
DoubleValue d; nodePtr->GetAttribute (
  “/NodeList/5/NetDevice/3/$ns3::WifiNetDevice/Phy
  /$ns3::YansWifiPhy/TxGain”, d);
```

- Users can get Ptrs to instances also, and Ptrs to trace sources, in the same way

# Attribute documentation



## The list of all attributes.

**[Core]**

Collaboration diagram for The list of all attributes.:

**ns3::V4Ping**

- Remote: The address of the machine we want to ping.

**ns3::ConstantRateWifiManager**

- DataMode: The transmission mode to use for every data packet transmission
- ControlMode: The transmission mode to use for every control packet transmission.

**ns3::WifiRemoteStationManager**

- IsLowLatency: If true, we attempt to modelize a so-called low-latency device: a device where decisions about tx parameters can be made on a per-packet basis and feedback about the transmission of each packet is obtained before sending the next. Otherwise, we modelize a high-latency device, that is a device where we cannot update our decision about tx parameters after every packet transmission.
- MaxSsrc: The maximum number of retransmission attempts for an RTS. This value will not have any effect on some rate control algorithms.
- MaxSlrc: The maximum number of retransmission attempts for a DATA packet. This value will not have any effect on some rate control algorithms.
- RtsCtsThreshold: If a data packet is bigger than this value, we use an RTS/CTS handshake before sending the data. This value will not have any effect on some rate control algorithms.

# Options to manipulate attributes

- Individual object attributes often derive from default values
  - Setting the default value will affect all subsequently created objects
  - Ability to configure attributes on a per-object basis
- Set the default value of an attribute from the command-line:

```
CommandLine cmd;
cmd.Parse (argc, argv);
```

- Set the default value of an attribute with NS_ATTRIBUTE_DEFAULT
- Set the default value of an attribute in C++:

```
Config::SetDefault ("ns3::Ipv4L3Protocol::CalcChecksum",
BooleanValue (true));
```

- Set an attribute directly on a specic object:

```
Ptr<CsmaChannel> csmaChannel = ...;
csmaChannel->SetAttribute ("DataRate",
StringValue ("5Mbps"));
```

# Summary on ns-3 objects

- ns-3 objects that inherit from base class ns3::Object get several additional features
  1. smart-pointer memory management (Class Ptr)
  2. dynamic run-time object aggregation
  3. an attribute system

- These types of objects are allocated on the heap, not on the stack

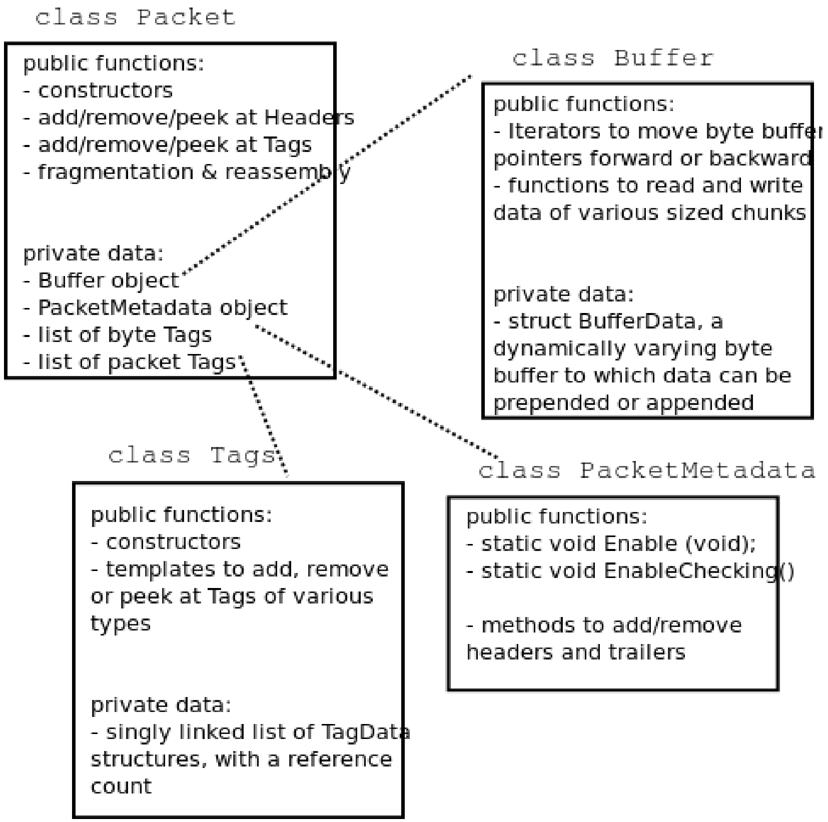
# Packets

# ns-3 Packet

- Packet is an advanced data structure with the following capabilities
  - Supports fragmentation and reassembly
  - Supports real or virtual application data
  - Extensible
  - Serializable (for emulation)
  - Supports pretty-printing
  - Efficient (copy-on-write semantics)

# ns-3 Packet structure

- Analogous to an mbuf/skbuff



`class Packet`

public functions:
- constructors
- add/remove/peek at Headers
- add/remove/peek at Tags
- fragmentation & reassembly

private data:
- Buffer object
- PacketMetadata object
- list of byte Tags
- list of packet Tags

`class Buffer`

public functions:
- Iterators to move byte buffer pointers forward or backward
- functions to read and write data of various sized chunks

private data:
- struct BufferData, a dynamically varying byte buffer to which data can be prepended or appended

`class Tags`

public functions:
- constructors
- templates to add, remove or peek at Tags of various types

private data:
- singly linked list of TagData structures, with a reference count

`class PacketMetadata`

public functions:
- static void Enable (void);
- static void EnableChecking()

- methods to add/remove headers and trailers

# Copy-on-write

- Copy data bytes only as needed

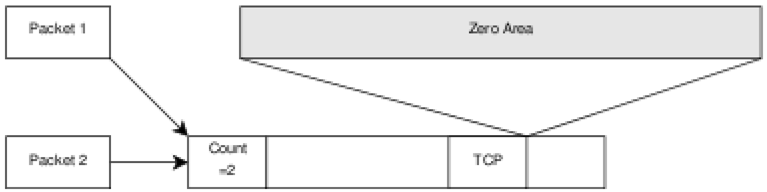


Figure 3.8: The TCP and the IP stacks hold references to a shared buffer.

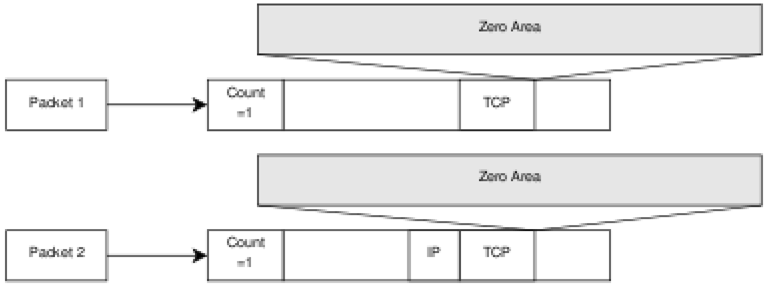


Figure 3.9: The IP stack inserts the IP header, triggers an un-share operation, completes the insertion.

Figure source: Mathieu Lacage's Ph.D. thesis

# Headers and trailers

- Most operations on packet involve adding and removing an ns3::Header
- class ns3::Header must implement four methods:

```
Serialize()

Deserialize()

GetSerializedSize()

Print()
```

# Headers and trailers (cont.)

- Headers are serialized into the packet byte buffer with Packet::AddHeader() and removed with Packet::RemoveHeader()
- Headers can also be 'Peeked' without removal

```
Ptr<Packet> pkt = Create<Packet> ();
UdpHeader hdr; // Note:  not heap allocated
pkt->AddHeader (hdr);
Ipv4Header iphdr;
pkt->AddHeader (iphdr);
```

# Packet tags

- Packet tag objects allow packets to carry around simulator-specific metadata
  - Such as a "unique ID" for packets or
- Tags may associate with byte ranges of data, or with the whole packet
  - Distinction is important when packets are fragmented and reassembled

# Tracing and statistics

- Tracing is a structured form of simulation output
- Example (from ns-2):

```
+ 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
- 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
r 1.84471 2 1 cbr 210 ------- 1 3.0 1.0 195 600
r 1.84566 2 0 ack 40 ------- 2 3.2 0.1 82 602
+ 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
```

Problem:  Tracing needs vary widely

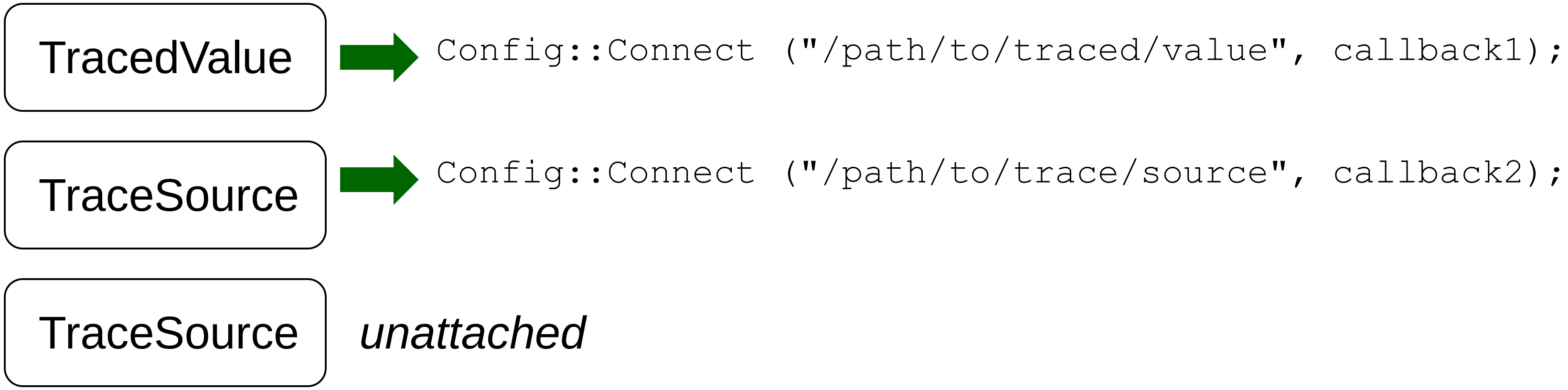
  - would like to change tracing output without editing the core
  - would like to support multiple outputs

# Tracing overview

- Simulator provides a set of pre-configured trace sources
  - Users may edit the core to add their own
- Users provide trace sinks and attach to the trace source
  - Simulator core provides a few examples for common cases
- Multiple trace sources can connect to a trace sink

# Tracing in ns-3

- ns-3 configures multiple 'TraceSource' objects (TracedValue, TracedCallback)
- Multiple types of 'TraceSink' objects can be hooked to these sources
- A special configuration namespace helps to manage access to trace sources



TracedValue

```
Config::Connect ("/path/to/traced/value", callback1);
```

TraceSource

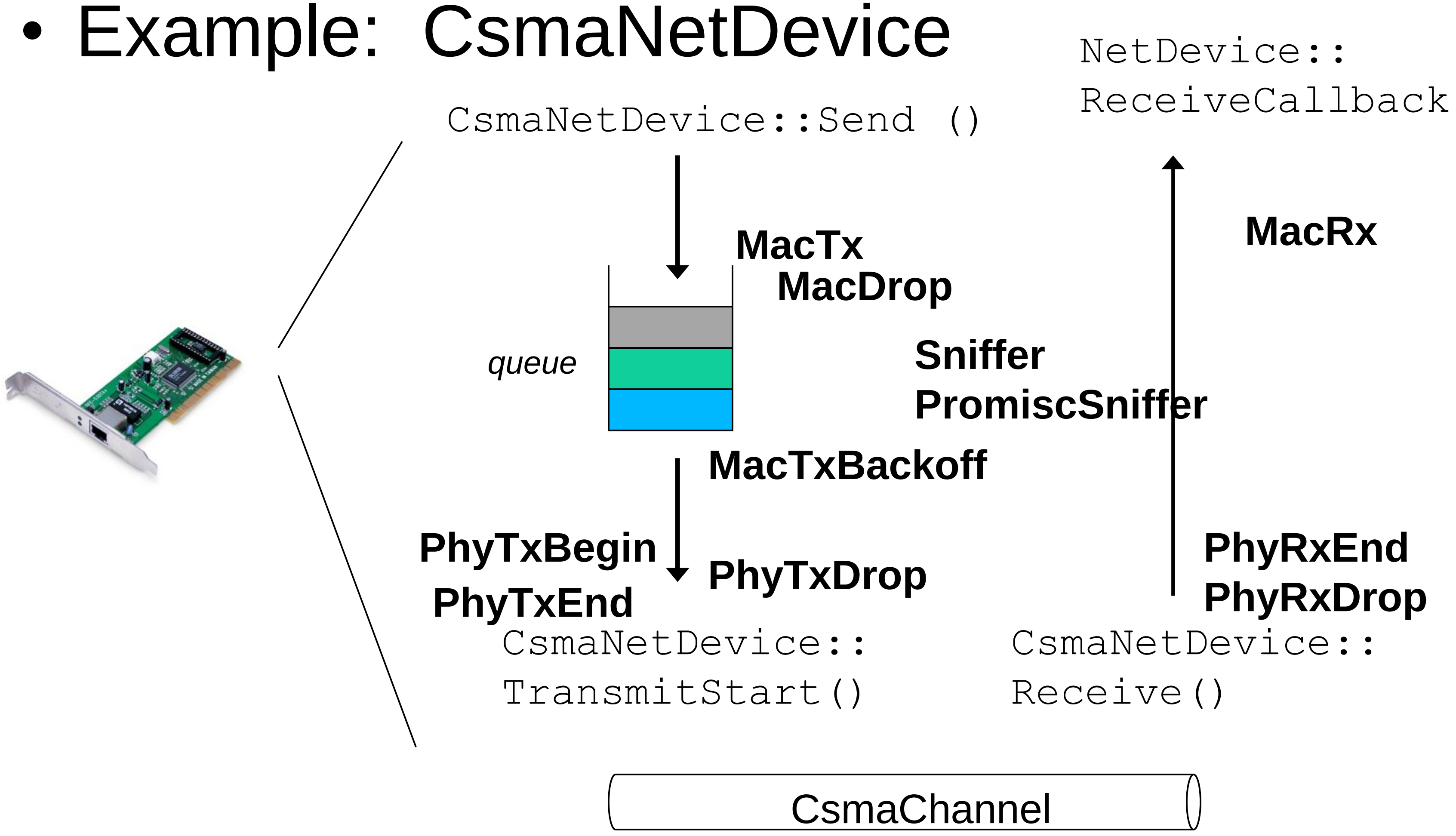
```
Config::Connect ("/path/to/trace/source", callback2);
```

TraceSource

*unattached*

# NetDevice trace hooks

- Example: CsmaNetDevice



`NetDevice::`
`ReceiveCallback`

`CsmaNetDevice::Send ()`

**MacRx**

**MacTx**

**MacDrop**

*queue*

**Sniffer**

**PromiscSniffer**

**MacTxBackoff**

**PhyTxBegin**

**PhyTxEnd**

**PhyTxDrop**

**PhyRxEnd**

**PhyRxDrop**

`CsmaNetDevice::`
`TransmitStart()`

`CsmaNetDevice::`
`Receive()`

CsmaChannel

# Writing and debugging your own examples

# Writing and debugging new programs

- Choosing between Python and C++
- Reading existing code
- Understanding and controlling logging code
- Error conditions
- Running programs through a debugger

# Python bindings

- ns-3 uses the 'pybindgen' tool to generate Python bindings for the underlying C++ libraries
- Existing bindings are typically found in the bindings/ directory of a module
- Some methods are not provided in Python (e.g. hooking trace sources)
- Generating new bindings requires a toolchain documented on the ns-3 web site

# Debugging support

- Assertions: NS_ASSERT (expression);
  - Aborts the program if expression evaluates to false
  - Includes source file name and line number
- Unconditional Breakpoints: NS_BREAKPOINT ();
  - Forces an unconditional breakpoint, compiled in
- Debug Logging (not to be confused with tracing!)
  - Purpose
    - Used to trace code execution logic
    - For debugging, not to extract results!
  - Properties
    - NS_LOG* macros work with C++ IO streams
    - E.g.: NS_LOG_UNCOND (”I have received ” << p->GetSize () << ” bytes”);
    - NS_LOG macros evaluate to nothing in optimized builds
    - When debugging is done, logging does not get in the way of execution performance

# Debugging support (cont.)

- Logging levels:
  - NS_LOG_ERROR (...): serious error messages only
  - NS_LOG_WARN (...): warning messages
  - NS_LOG_DEBUG (...): rare ad-hoc debug messages
  - NS_LOG_INFO (...): informational messages (eg. banners)
  - NS_LOG_FUNCTION (...):function tracing
  - NS_LOG_PARAM (...): parameters to functions
  - NS_LOG_LOGIC (...): control flow tracing within functions
- Logging ”components”
  - Logging messages organized by components
  - Usually one component is one .cc source file
  - NS_LOG_COMPONENT_DEFINE ("OlsrAgent");
- Displaying log messages. Two ways:
  - Programatically:
    - LogComponentEnable("OlsrAgent", LOG_LEVEL_ALL);
  - From the environment:
    - NS_LOG="OlsrAgent" ./my-program

# Running C++ programs through gdb

- The gdb debugger can be used directly on binaries in the build directory
- An easier way is to use a waf shortcut

```
./waf --command-template="gdb %s" --run <program-
  name>
```

# Running C++ programs through valgrind

- valgrind memcheck can be used directly on binaries in the build directory
- An easier way is to use a waf shortcut

```
./waf --command-template="valgrind %s" --run
  <program-name>
```

- Note: disable GTK at configure time when running valgrind (to suppress spurious reports)
- `./waf configure --disable-gtk --enable-tests ...`

# Testing

- Can you trust ns-3 simulations?
  - Can you trust *any* simulation?
    - Onus is on the simulation project to validate and document results
    - Onus is also on the researcher to verify results
- ns-3 strategies:
  - regression tests
    - Aim for ***event-based*** rather than ***trace-based***
  - unit tests for verification
  - validation of models on testbeds where possible
  - reuse of code

# Test framework

- ns-3-dev is checked nightly on multiple platforms
  - Linux gcc-4.x, i386 and x86_64, OS X, FreeBSD clang, and Cygwin (occasionally)
- `./test.py` will run regression tests

Walk through test code, test terminology (suite, case), and examples of how tests are run

# Improving performance

- Debug vs optimized builds
  - ./waf -d debug configure
  - ./waf -d debug optimized

- Build ns-3 with static libraries
  - ./waf --enable-static

- Use different compilers (icc)
  - has been done in past, not regularly tested