An ns-3 tutorial

Mathieu Lacage
mathieu.lacage@sophia.inria.fr

INRIA

Tunis, April, 7th and 8th 2009

Goals of this tutorial

- Learn about the ns-3 project and its goals
- Understand the software architecture, conventions, and basic usage of ns-3
- Read and modify an example ns-3 script
- Learn how you might extend ns-3 to conduct your own research
- Provide feedback to the ns-3 development team

Assumptions

Some familiarity with:

- C++ and Python programming language
- TCP/IP
- Unix Network Programming (e.g., sockets)
- Discrete-event network simulation

Tutorial schedule

Tuesday 7th:

- 14h00-16h00: Overview of ns-3 features
- 16h00-16h30: Pause
- 16h30-17h30: An end-to-end tour of a simulation
- 17h30-18h00: Experimentation

Wednesday 8th:

- 09h00-10h00: Setup
- ...

Part I

Overview of ns-3 features

Outline

The basics

- ns-3 is written in C++
- Bindings in Python
- ns-3 uses the waf build system
- simulation programs are C++ executables or python scripts
- API documentation using doxygen

The waf build system

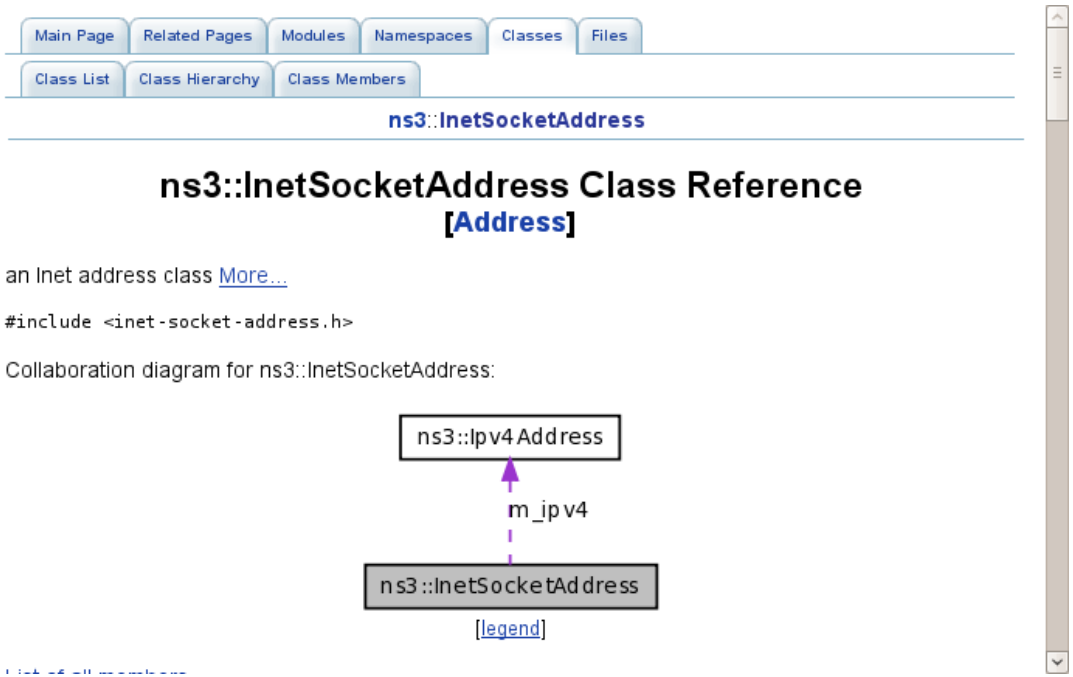
Waf is a Python-based framework for configuring, compiling and installing applications: a replacement for other tools such as Autotools, Scons, CMake or Ant.

- Homepage with documentation: `http://code.google.com/p/waf/`
- For those familiar with autotools:
  - ./configure → ./waf -d [optimized—debug] configure
  - make → ./waf
  - make test → ./waf check
- Can run programs through a special waf shell: this gets the library paths right for you:
  - ./waf –run simple-point-to-point
  - ./waf –shell

APIs

We use Doxygen:

- last stable release: `http://www.nsnam.org/doxygen-release/index.html`
- development tree: `http://www.nsnam.org/doxygen/index.html`



Getting started I

Install all needed tools:

Ubuntu

```
sudo apt-get install build-essential g++ python mercurial
```

Windows

- cygwin
- python
- mercurial

Getting started II

Download and build development version:

```
hg clone http://code.nsnam.org/ns-3-allinone
cd ns-3-allinone
./download.py
./build.py
cd ns-3-dev
./waf distclean
./waf configure
./waf
```

Useful options:

- `./waf -d optimized configure`
- `-j#` where # is number of cores
- `./waf --help` shows you other options

Getting started III

Running programs:

- build/VARIANT:
  - Programs compile against build/VARIANT/ns3/*.h
  - Programs link against shared library build/VARIANT/libns3.so
  - Programs are built as build/VARIANT/PATH/program-name
- Using `./waf --shell`

```
./waf --shell
./build/debug/samples/main-simulator
```

- Using `./waf --run`

```
./waf --run examples/csma-bridge.cc
./waf --pyrun examples/csma-bridge.py
```

Simulation basics

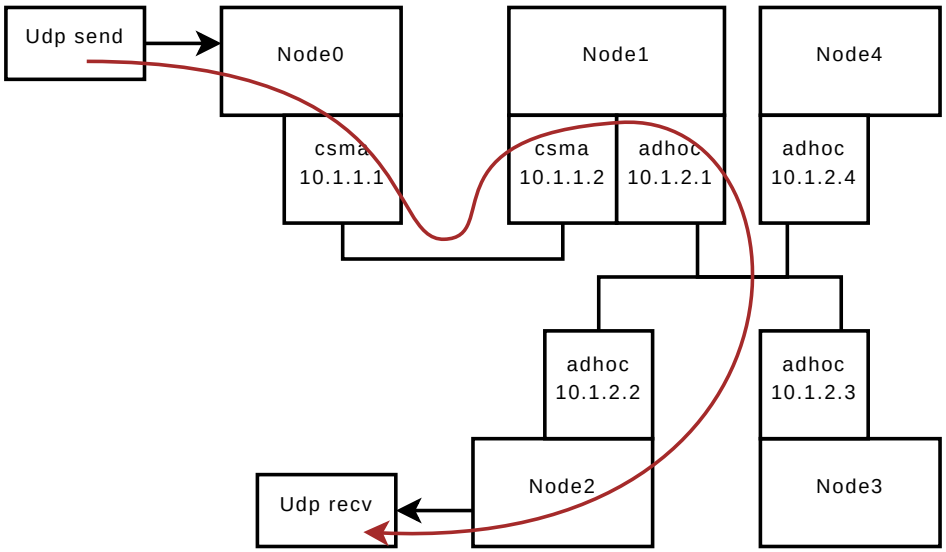
- Simulation time moves discretely from event to event
- C++ functions schedule events to occur at specific simulation times
- A simulation scheduler orders the event execution
- `Simulation::Run` gets it all started
- Simulation stops at specific time or when events end

Scheduling events

In samples/main-simulation.cc:

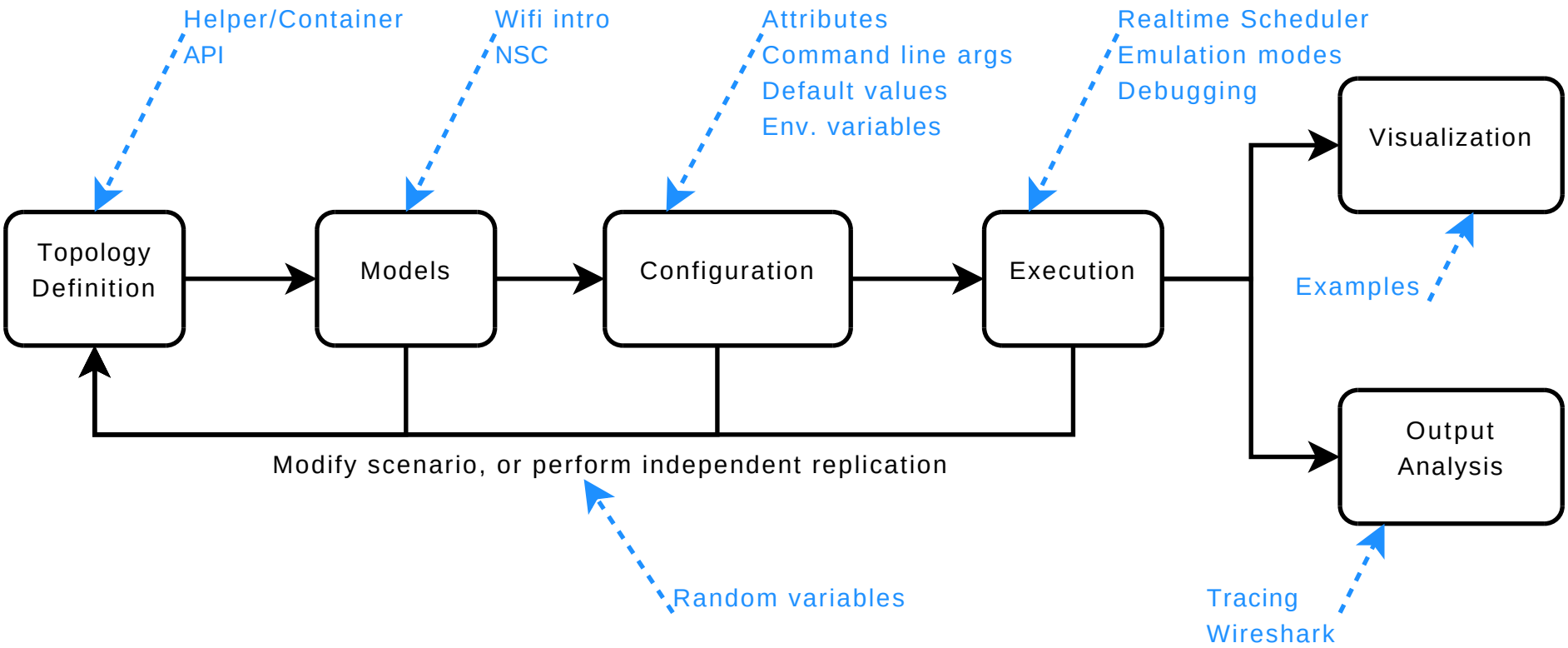
```
static void random_function (MyModel *model)
{
  // print some stuff
}
int main (int argc, char **argv)
{
  MyModel model;
  Simulator::Schedule (Seconds (10.0),
                       &random_function, &model);
  Simulator::Run ();
  Simulator::Destroy ();
}
```

The Testcase

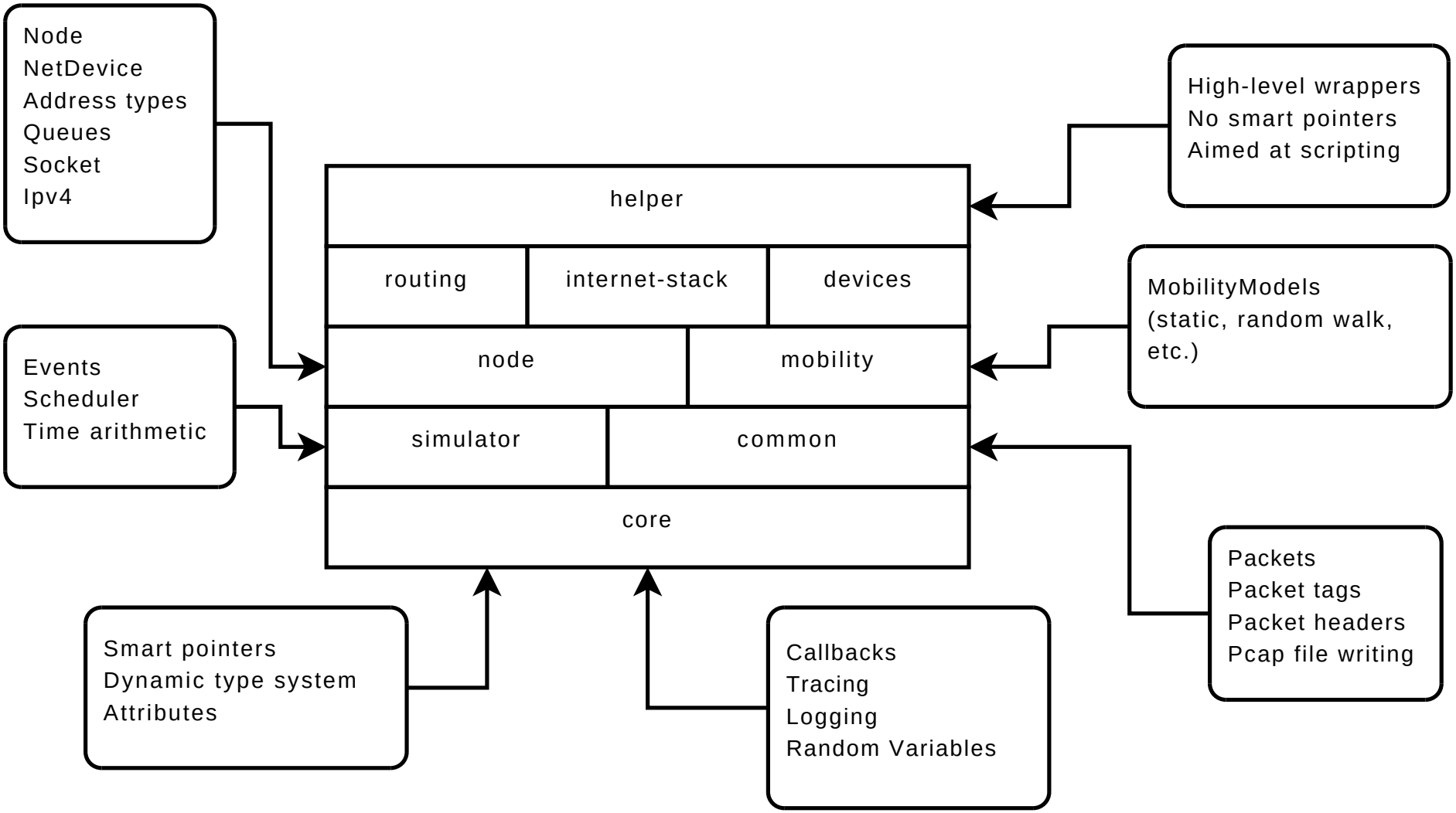


- One csma link
- One wifi infrastructure network
- Two ip subnetworks
- One udp traffic generator
- One udp traffic receiver
- Global god ip routing

A typical simulation structure



Where is everything located ?



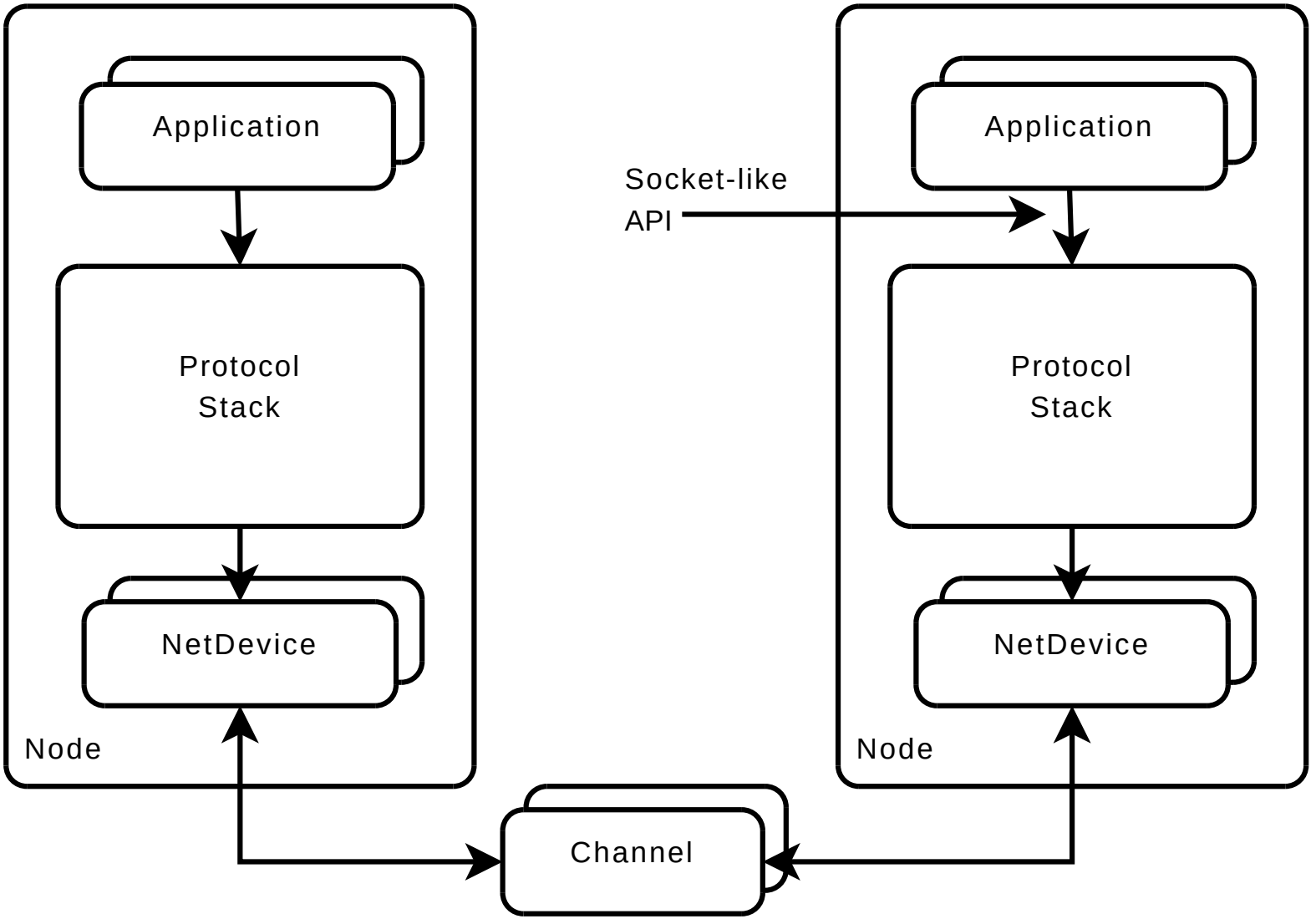
So, what does the source code look like ?

Let's open `tutorial-helper.cc`

Topology construction

```
NodeContainer csmaNodes;
csmaNodes.Create (2);
...
NetDeviceContainer csmaDevices;
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate",
                          StringValue ("5Mbps"));
csma.SetChannelAttribute ("Delay",
                          StringValue ("2ms"));
csmaDevices = csma.Install (csmaNodes);
```

The basic model

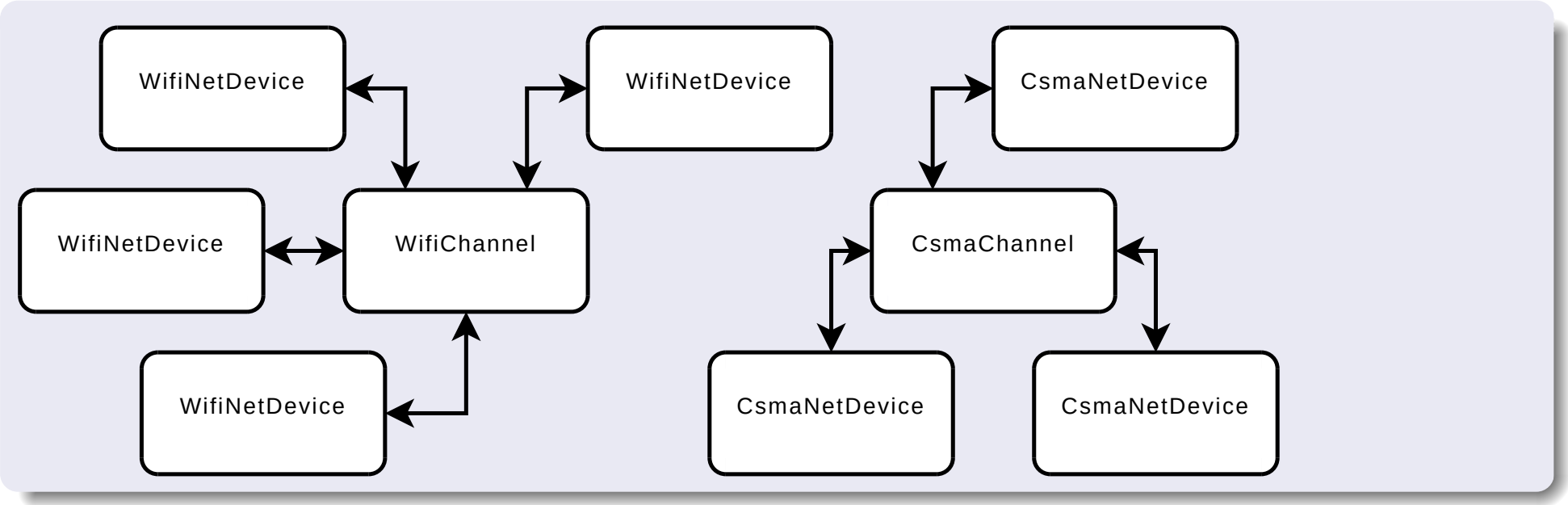


The fundamental objects

- Node: the motherboard of a computer with RAM, CPU, and, IO interfaces
- Application: a packet generator and consumer which can run on a Node and talk to a set of network *stacks*
- NetDevice: a network card which can be plugged in an IO interface of a Node
- Channel: a physical connector between a set of NetDevice objects

Important remarks

- NetDevices are strongly bound to Channels of a matching type:



- Nodes are architected for multiple interfaces

Existing model implementations

- Network stacks: ipv4, icmpv4, udp, tcp (ipv6 under review)
- Devices: wifi, csma, point-to-point, bridge
- Error models and queues
- Applications: udp echo, on/off, sink
- Mobility models: random walk, etc.
- Routing: olsr, static global

The helper/container API I

We want to:

- Make it easy to build topologies with repeating patterns
- Make the topology description more high-level (and less verbose) to make it easier to read and understand

The idea is simple:

- Sets of objects are stored in `Containers`
- One operation is encoded in a `Helper` object and applies on a `Container`

Helper operations:

- Are not generic: different helpers provide different operations
- Do not try to allow code reuse: just try to minimize the amount of code written
- Provide *syntactical sugar*: make the code easier to read

The helper/container API II

Example containers:

- `NodeContainer`
- `NetDeviceContainer`
- `Ipv4AddressContainer`

Example helper classes:

- `InternetStackHelper`
- `WifiHelper`
- `MobilityHelper`
- `OlsrHelper`
- etc. Each model provides a helper class

The helper/container API III

For example, we create a couple of nodes:

```
NodeContainer csmaNodes;
csmaNodes.Create (2);
NodeContainer wifiNodes;
wifiNodes.Add (csmaNodes.Get (1));
wifiNodes.Create (3);
```

Create empty node container
Create two nodes
Create empty node container
Add existing node to it
And then create some more nodes

And, then, we create the csma network:

```
NetDeviceContainer csmaDevices;
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate",
                          StringValue ("5Mbps"));
csma.SetChannelAttribute ("Delay",
                          StringValue ("2ms"));
csmaDevices = csma.Install (csmaNodes);
```

Create empty device container
Create csma helper
Set data rate

Set delay

Create csma devices and channel

The helper/container API IV

Finally, setup the wifi channel:

```
YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default ();
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
wifiPhy.SetChannel (wifiChannel.Create ());
```

And create adhoc devices on this channel:

```
NetDeviceContainer wifiDevices;
WifiHelper wifi = WifiHelper::Default ();
wifiDevices = wifi.Install (wifiPhy, wifiNodes);
```

The wifi models

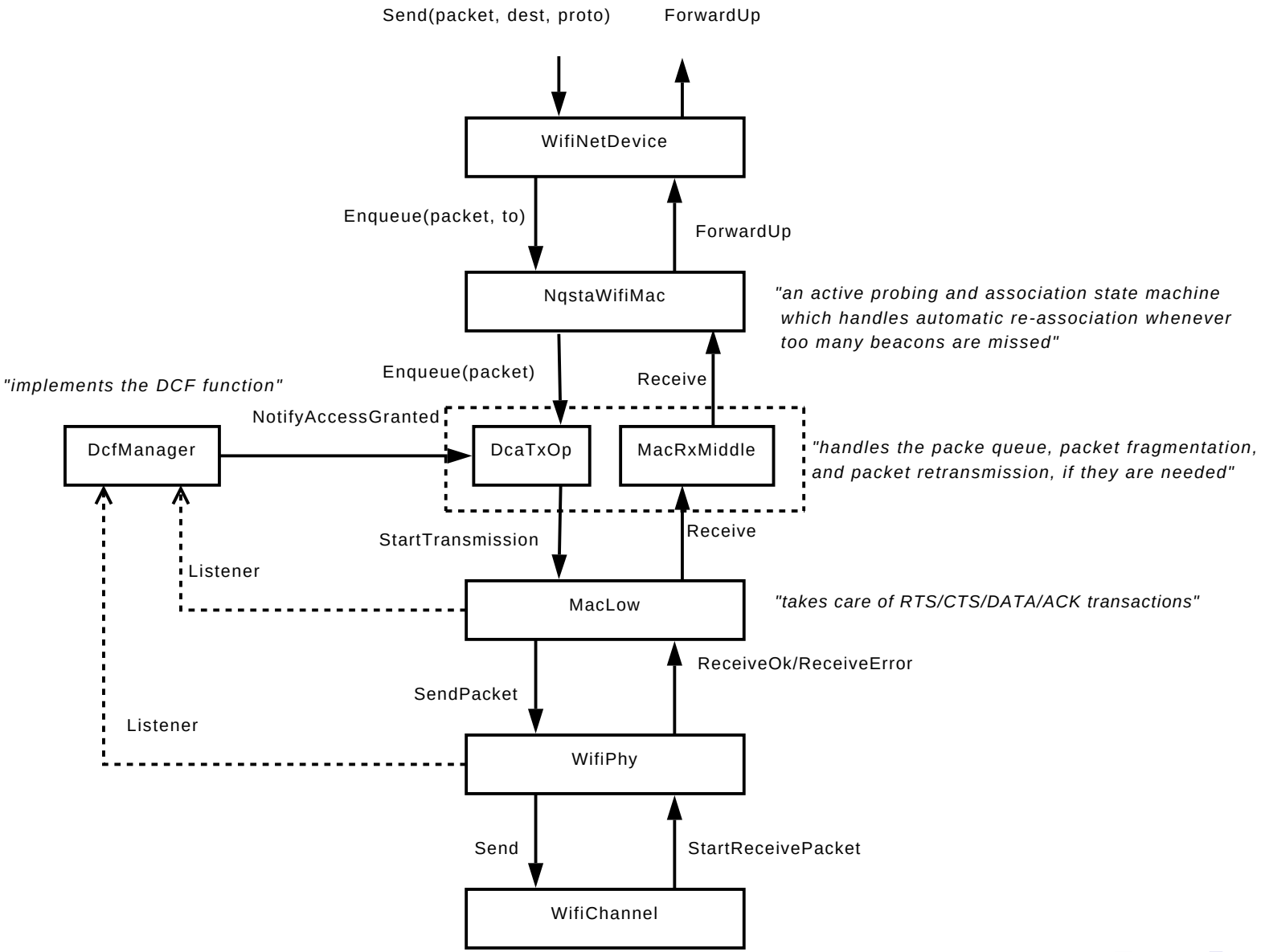
- New model, written from 802.11 specification
- Accurate model of the MAC
- DCF, beacon generation, probing, association
- A set of rate control algorithms (ARF, ideal, AARF, etc.)
- Not-so-slow models of the 802.11a PHY

Development of wifi models

New contributions from many developers:

- University of Florence: 802.11n, EDCA, frame aggregation, block ack
- Russian Academy of Sciences: 802.11s, HWMP routing protocol
- Boeing: 802.11b channel models, validation
- Deutsche Telekom Laboratories: PHY modelization, validation
- Karlsruhe Institute of Technology: PHY modelization (Rayleigh, Nakagami)

Wifi implementation



Mobility models I

Setup the initial position:

```
MobilityHelper mobility;
mobility.SetPositionAllocator ("ns3::RandomDiscPositionAllocator",
                               "X", StringValue ("100.0"),
                               "Y", StringValue ("100.0"),
                               "Rho", StringValue ("Uniform:0:30"));
```

And setup the mobility model during simulation:

```
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiNodes);
```

Mobility models II

- The `MobilityModel` base class:
  - `SetPosition`, `GetPosition`: 3d x,y,z vector
  - `GetVelocity`: 3d x,y,z vector
- `ConstantPositionMobilityModel`: node is at a fixed location; does not move on its own
- `ConstantVelocityMobilityModel`: node moves with a fixed velocity vector
- `ConstantAccelerationMobilityModel`: node moves with a fixed acceleration vector
- `RandomWaypointMobilityModel`:
  - Node pauses for a certain random time
  - Node selects a random waypoint and speed
  - Node starts walking towards the waypoint
  - When waypoint is reached, goto first state

Mobility models III

- `RandomDirection2dMobilityModel`
  - Works inside a rectangular bounded area
  - Node selects a random direction and speed
  - Node walks in that direction until the edge
  - Node pauses for random time
  - Repeat
- `RandomWalk2dMobilityModel`: brownian motion
  - Works inside a rectangular bounded area
  - Node selects a random direction and speed
  - Node walks in that direction until either a fixed delay, or distance expires, or edge is hit
  - If edge is hit, reflexive angle, same speed
  - If fixed distance or delay expires, repeat random direction and speed
- `HierarchicalMobilityModel`: for group mobility and complex mobility patterns
- `Ns2MobilityHelper`: for ns-2 mobility files

Ipv4, Udp, Tcp models I

Install the ipv4, udp, and tcp stacks:

```
InternetStackHelper internet;
internet.Install (NodeContainer::GetGlobal ());
```

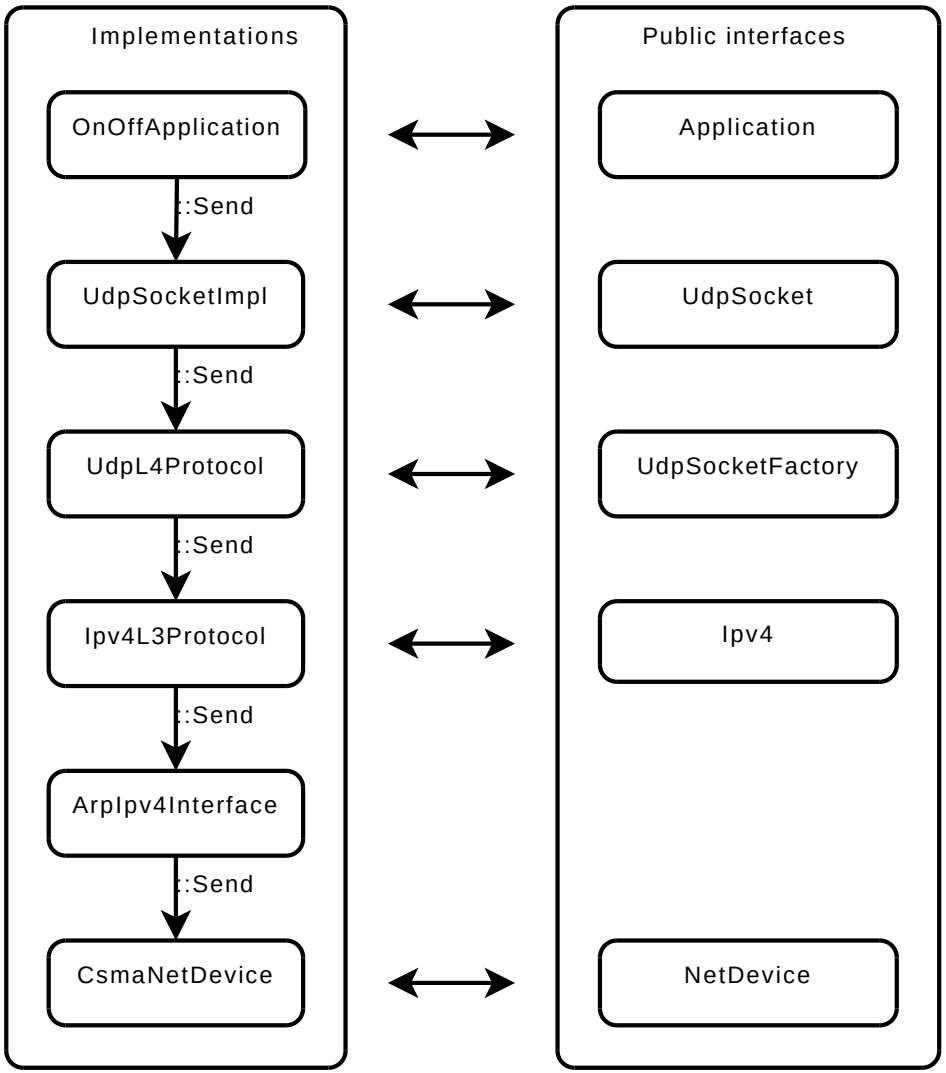
Assign ipv4 addresses:

```
Ipv4InterfaceContainer csmaInterfaces;
Ipv4InterfaceContainer wifiInterfaces;
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
csmaInterfaces = ipv4.Assign (csmaDevices);
ipv4.SetBase ("10.1.2.0", "255.255.255.0");
wifiInterfaces = ipv4.Assign (wifiDevices);
```

Setup routing tables:

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

Internet Stack Architecture



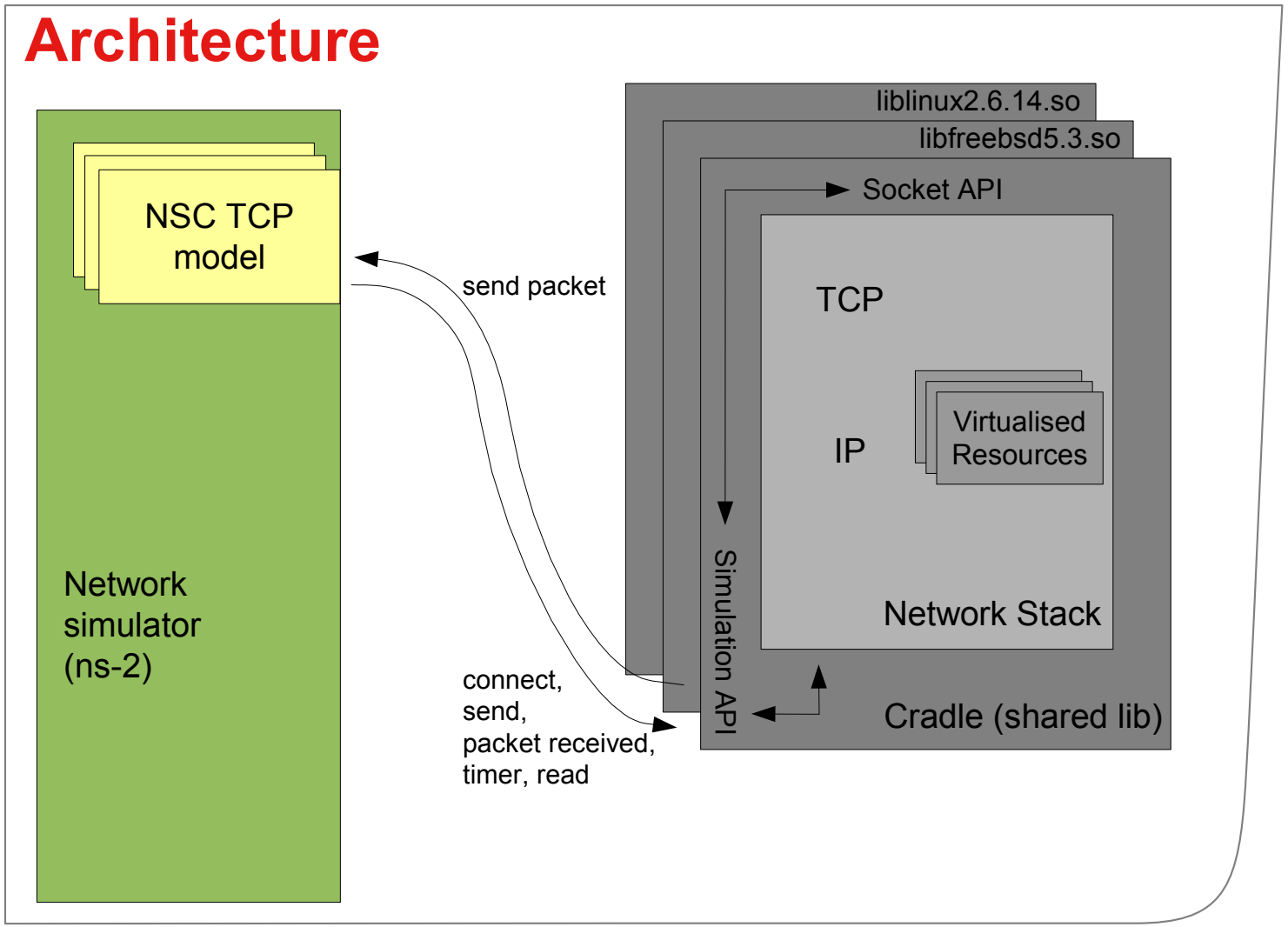
Multiple TCP implementations

- Native ns-3 TCP (ported from GTNetS)
- Network Simulation Craddle: linux, BSD, etc.

To enable NSC:

```
internetStack.SetNscStack ("liblinux2.6.26.so");
```

NSC Architecture



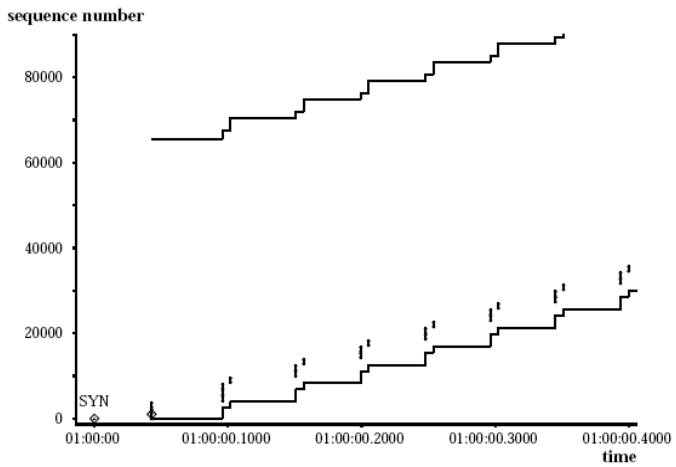
NSC implementation

- Globalizer: per-process kernel source code and add indirection to all global variable declarations and accesses
- Glue: per-kernel (and per-kernel-version) glue to provide kernel APIs for kernel code:
  - `kmalloc`: memory allocation
  - `NetDevice` integration
  - `Socket` integration
- Provides glue for:
  - linux 2.6.18, 2.6.26, 2.6.28
  - FreeBSD 5
  - lwip 1.3
  - OpenBSD 3

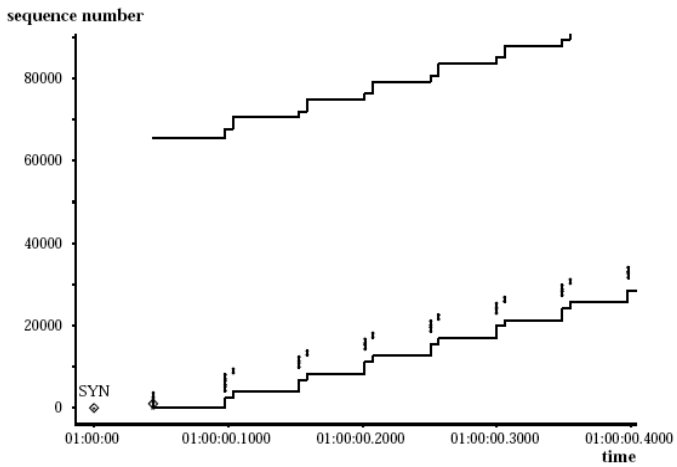
Nsc accuracy

Accuracy

•Have shown NSC to be very accurate – able to produce packet traces that are almost identical to traces measured from a test network



(a) Simulated FreeBSD



(b) Measured FreeBSD

Ipv4 Routing

The base class: `Ipv4RoutingProtocol`. Multiple implementations:

- Olsr
  - Dynamic routing
  - Can handle wireless and wired topologies
- Static:
  - Used by global routing
  - Mainly for static topologies
  - Works well for csma and point to point links

In the future: integrate Quagga, Xorp.

Traffic generation and reception I

Prepare to create an on-off traffic generator:

```
OnOffHelper onoff ("ns3::UdpSocketFactory",
                   InetSocketAddress ("10.1.2.2", 1025));
onoff.SetAttribute ("OnTime", StringValue ("Constant:1.0"));
onoff.SetAttribute ("OffTime", StringValue ("Constant:0.0"));
```

Install the traffic generator:

```
ApplicationContainer apps;
apps = onoff.Install (csmaNodes.Get (0));
```

and start it:

```
apps.Start (Seconds (1.0));
apps.Stop (Seconds (4.0));
```

Traffic generation and reception II

Setup the traffic sink:

```
PacketSinkHelper sink ("ns3::UdpSocketFactory",
                       InetSocketAddress ("10.1.2.2", 1025));
apps = sink.Install (wifiNodes.Get (1));
apps.Start (Seconds (0.0));
apps.Stop (Seconds (4.0));
```

Traffic generation and reception III

Applications:

- All subclass `ns3::Application`
- Are managed by a Node
- Represent a process on an actual system
- Talk to network stacks through one or more sockets

ns-3 Sockets

Plain C sockets:

```
int sk;
sk = socket(PF_INET, SOCK_DGRAM, 0);

struct sockaddr_in src;
inet_pton(AF_INET,"0.0.0.0",&src.sin_addr);
src.sin_port = htons(80);
bind(sk, (struct sockaddr *) &src,sizeof(src));

struct sockaddr_in dest;
inet_pton(AF_INET,"10.0.0.1",&dest.sin_addr);
dest.sin_port = htons(80);
connect(sk,(struct sockaddr *) &dest,
        sizeof(dest));

send(sk, "hello", 6, 0);




char buf[6];
recv(sk, buf, 6, 0);
```

ns-3 sockets:

```
Ptr<Socket> sk;
sk = udpFactory->CreateSocket ();




sk->Bind (InetSocketAddress (80));


Ipv4Address ipv4Dst ("10.0.0.1")
InetSocketAddress dst (ipv4Dst, 80)
sk->Connect(dst);


sk->Send (Create<Packet> ("hello", 6));

void MySocketReceive (Ptr<Socket> sk,
                      Ptr<Packet> packet)
{
...
}
sk->SetReceiveCallback (
          MakeCallback (MySocketReceive));
[...] Simulator::Run ()
```

Object attributes

Configure the default value of all attributes:

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

Setting individual attributes:

```
onoff.SetAttribute ("OnTime", StringValue ("Constant:1.0"));
onoff.SetAttribute ("OffTime", StringValue ("Constant:0.0"));
```

Configure and explore all attributes:

```
GtkConfigStore config;
config.ConfigureDefaults ();
...
config.ConfigureAttributes ();
```

What can attributes be used for ?

A researcher wants to:

- Discover all the ways he can tweak his models
- Save for each simulation the value of all control variables
- Tweak all control variables

In ns-3, attributes provide a unified way to do this:

- An attribute represents a value in an object
- We can visualize all of them in a GUI
- We can dump and read them all in configuration files
- We can set them all from the command-line
- We can set them all from environment variables

Traditionally, in C++:

- 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

For example:

```
class MyModel {
public:
  MyModel () : m_foo (m_defaultFoo) {}
  void SetFoo (int foo) {m_foo = foo;}
  int GetFoo (void) {return m_foo}
  static void SetDefaultFoo (int foo) {m_defaultFoo = foo;}
  static int GetDefaultFoo (void) {return m_defaultFoo;}
private:
  int m_foo;
  static int m_defaultFoo = 10;
};
```

In ns-3, it's all done automatically I

- Set a default value:

```
Config::SetDefaultValue ("ns3::WifiPhy::TxGain", StringValue ("10"));
```

- Set a value on a specific object:

```
phy->SetAttribute ("TxGain", StringValue ("10"));
```

- Set a value deep in the system using a namespace string:

```
Config::SetAttribute ("/NodeList/5/DeviceList/0/Phy/TxGain",
                      StringValue ("10"));
```

- Set a value from the command-line `--ns3::WifiPhy::TxGain=10`:

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

In ns-3, it's all done automatically II

All automatically:

- Load, Change, and Save all values from and to a raw text or xml file with a GUI:

```
GtkConfigStore config;
config.ConfigureDefaults ();
...
config.ConfigureAttributes ();
```

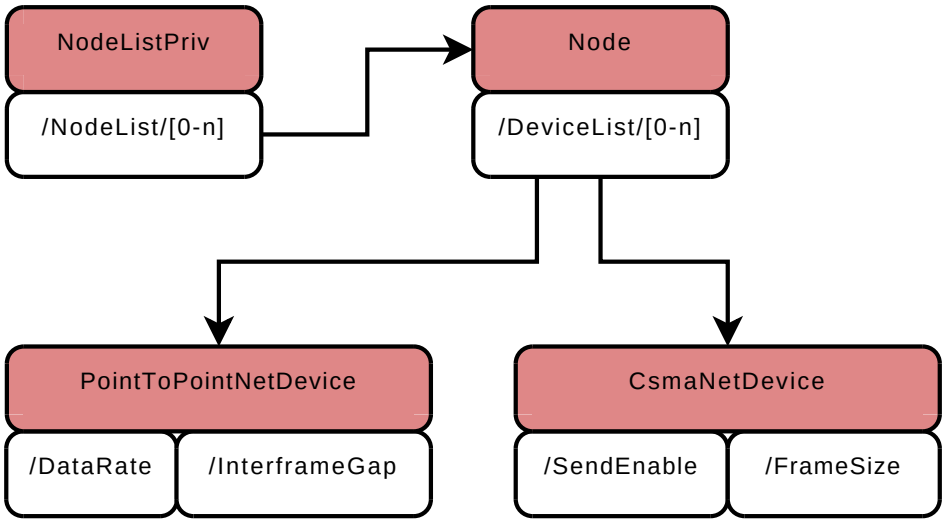
- Load and Save all values from and to a raw text or xml file:

```
ConfigStore config;
config.ConfigureDefaults ();
...
config.ConfigureAttributes ();
```

- Set a value with an environment variable
  `NS_ATTRIBUTE_DEFAULT=ns3::WifiPhy::TxGain=10`

The attribute namespace I

Attribute namespace strings represent a path through a set of object pointers:



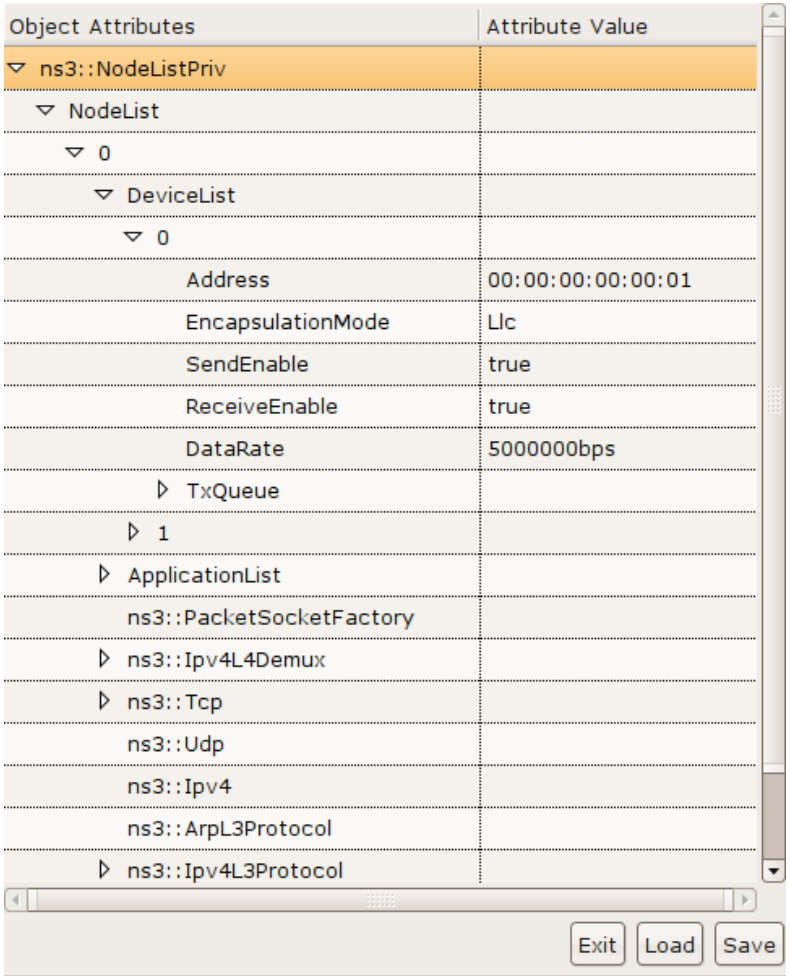
For example, `/NodeList/x/DeviceList/y/InterframeGap` represents the `InterframeGap` attribute of the device number y in node number x.

The attribute namespace II

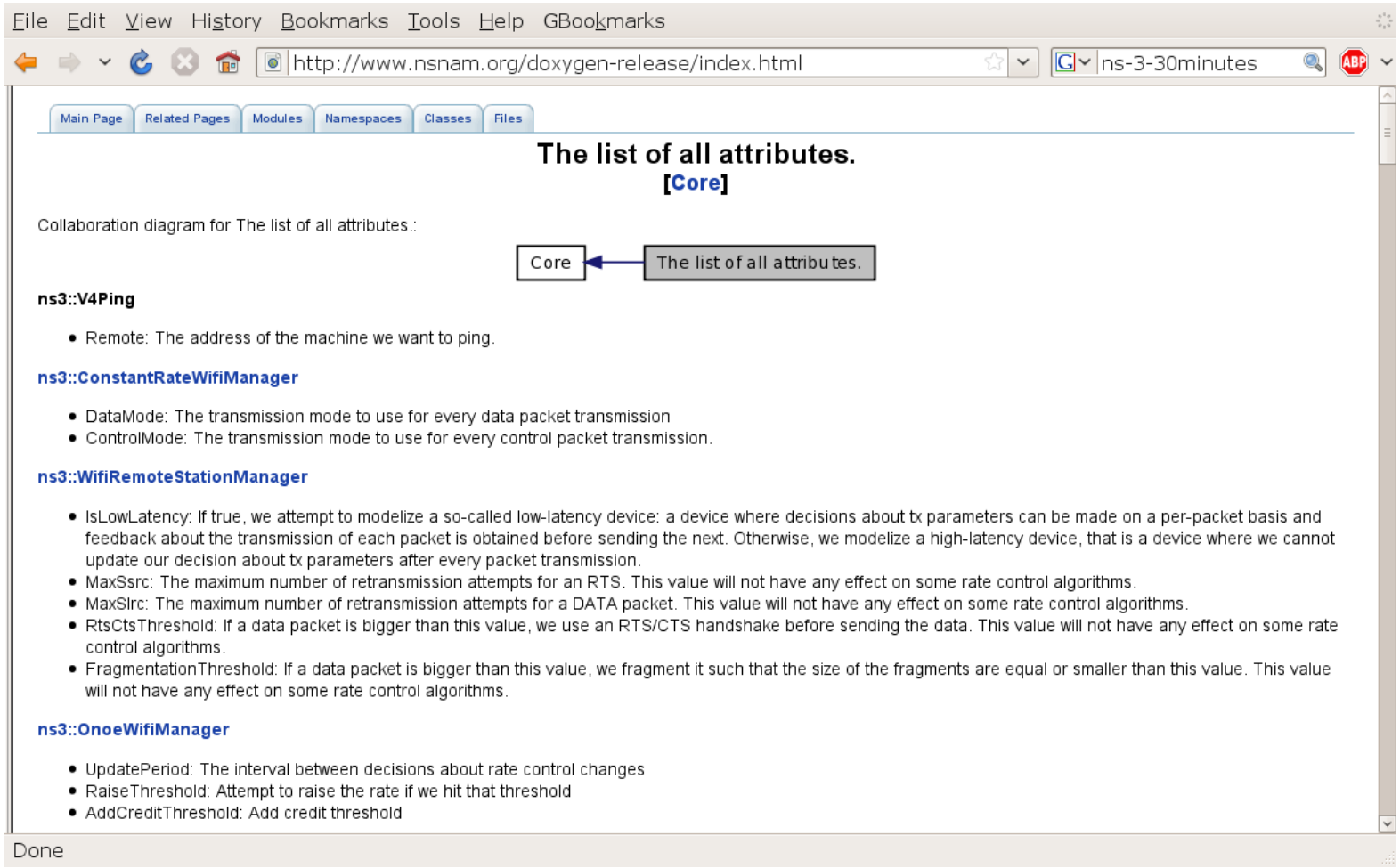
Navigating the attributes using paths:

- `/NodeList/[3-5]|8|[0-1]`: matches nodes index 0, 1, 3, 4, 5, 8
- `/NodeList/*`: matches all nodes
- `/NodeList/3/$ns3::Ipv4`: matches object of type `ns3::Ipv4` aggregated to node number 3
- `/NodeList/3/DeviceList/*/$ns3::CsmaNetDevice`: matches all devices of type `ns3::CsmaNetDevice` within node number 3

A graphical navigation of the attributes



Doxygen documentation of all attributes



Tracing

Setup simple ascii and pcap tracing:

```
std::ofstream ascii;
ascii.open ("wns3-helper.tr");
CsmaHelper::EnableAsciiAll (ascii);
CsmaHelper::EnablePcapAll ("wns3-helper");
YansWifiPhyHelper::EnablePcapAll ("wsn3-helper");
```

Tracing requirements

- 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
```

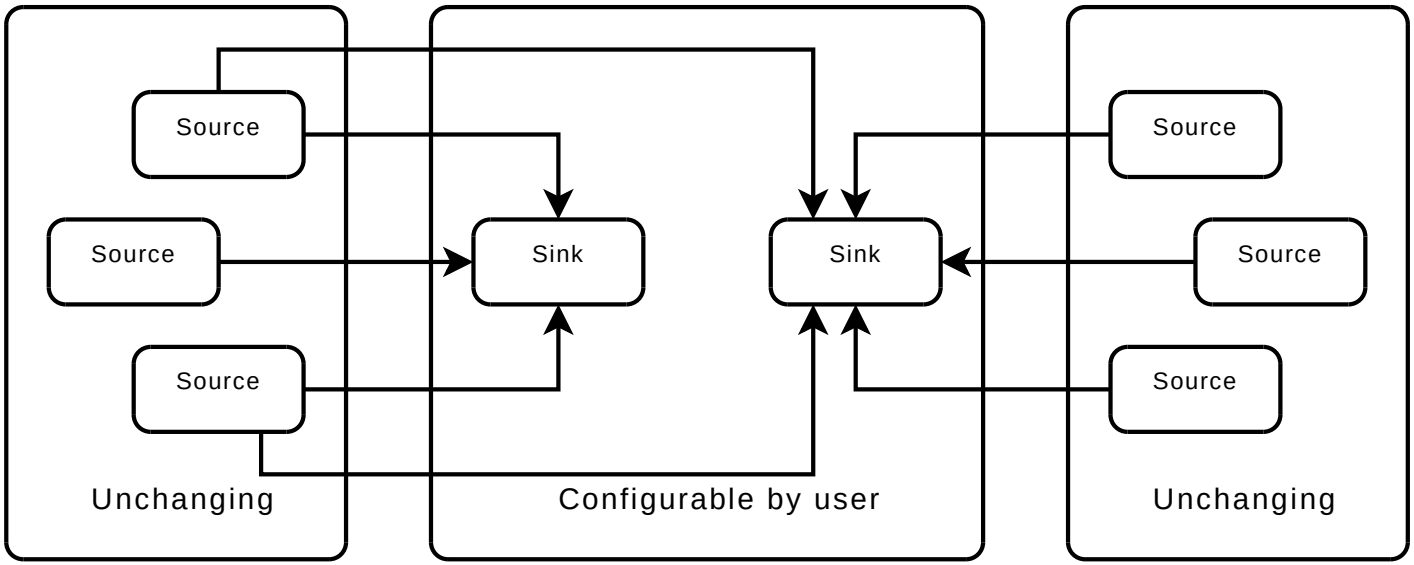
- Problem: tracing needs vary widely
  - Would like to change tracing output without editing the core
  - Would like to support multiple outputs

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

The ns-3 tracing model

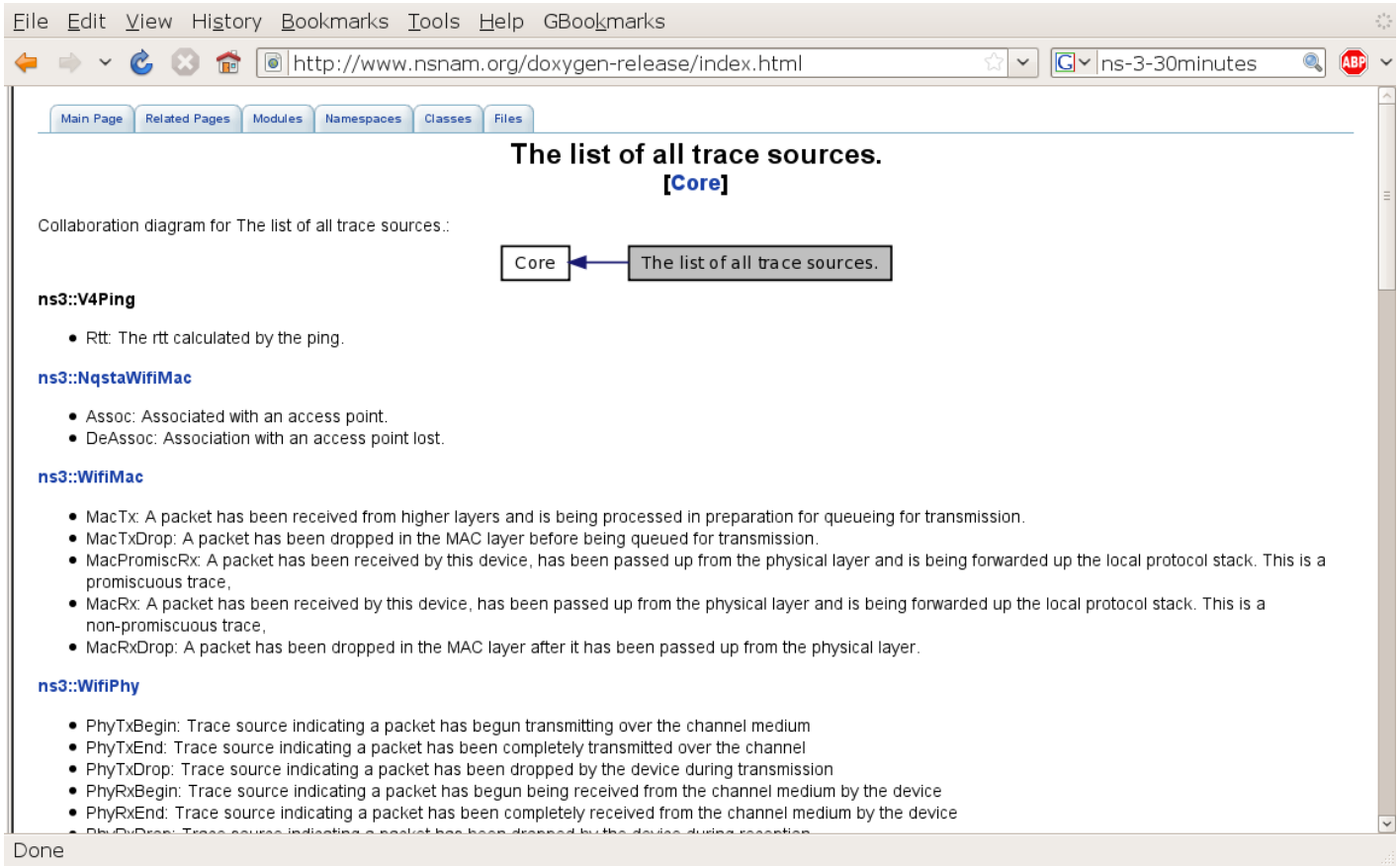
Decouple trace sources from trace sinks:



Benefit: Customizable trace sinks

Ns-3 trace sources

- Various trace sources (e.g., packet receptions, state machine transitions) are plumbed through the system
- Organized with the rest of the attribute system



Multiple levels of tracing

- High-level: use a helper to hook a predefined trace sink to a trace source and generate simple tracing output (ascii, pcap)
- Mid-level: hook a special trace sink to an existing trace source to generate adhoc tracing
- Low-level: add a new trace source and connect it to a special trace sink

High-level tracing

- Use predefined trace sinks in helpers
- All helpers provide ascii and pcap trace sinks

```
CsmaHelper::EnablePcap ("filename", nodeid, deviceid);
std::ofstream os;
os.open ("filename.tr");
CsmaHelper::EnableAscii (os, nodeid, deviceid);
```

Mid-level tracing

- Provide a new trace sink
- Use attribute/trace namespace to connect trace sink and source

```
void
DevTxTrace (std::string context,
            Ptr<const Packet> p, Mac48Address address)
{
  std::cout << " TX to=" << address << " p: " << *p << std::endl;
}
Config::Connect ("/NodeList/*/DeviceList/*/Mac/MacTx",
                 MakeCallback (&DevTxTrace));
```

Using mid-level tracing for pcap output

The trace sink:

```
static void PcapSnifferEvent (Ptr<PcapWriter> writer,
                              Ptr<const Packet> packet)
{
  writer->WritePacket (packet);
}
```

Prepare the pcap output:

```
oss << filename << "-" << nodeid << "-" << deviceid << ".pcap";
Ptr<PcapWriter> pcap = ::ns3::Create<PcapWriter> ();
pcap->Open (oss.str ());
pcap->WriteWifiHeader ();
```

Finally, connect the trace sink to the trace source:

```
oss << "/NodeList/" << nodeid << "/DeviceList/" << deviceid;
oss << "/$ns3::WifiNetDevice/Phy/PromiscSniffer";
Config::ConnectWithoutContext (oss.str (),
                MakeBoundCallback (&PcapSnifferEvent, pcap));
```

Random variables

Specify a *random* start time:

```
onoff.SetAttribute ("OnTime", StringValue ("Uniform:1.0:2.0"));
```

- Uses the MRG32k3a generator from Pierre L'Ecuyer (like ns-2):
  - `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/streams00.pdf`
  - Period of PRNG is $3.1x10^{57}$
- Partitions a pseudo-random number generator into uncorrelated streams and substreams:
  - Each RandomVariable gets its own stream
  - This stream partitioned into substreams
- If you increment the seed of the PRNG, the RandomVariable streams 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

Random variables in ns-3

Ns-3 simulations use a fixed seed (1) and run number (1) by default:

- `--RngSeed`, `--RngRun`
- `NS_GLOBAL_VALUE=RngRun=2`
- Default was random seeding prior to 3.4

To run independent replications of the same scenario:

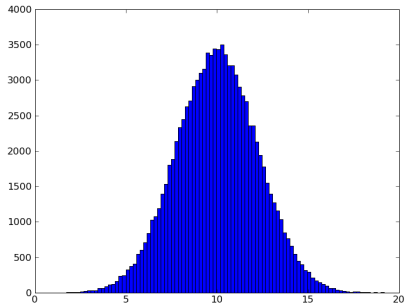
- Don't change the seed ever
- Increment the run number for each simulation

Random variable API

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,
- etc.

```
import pylab
import ns3
rng = ns3.NormalVariable(10.0, 5.0)
x = [rng.GetValue() for t in range(100000)]
pylab.hist(x,100)
pylab.show()
```



Pause !!

Let's have a break

Part II

An end-to-end tour of a simulation

Outline

Outline

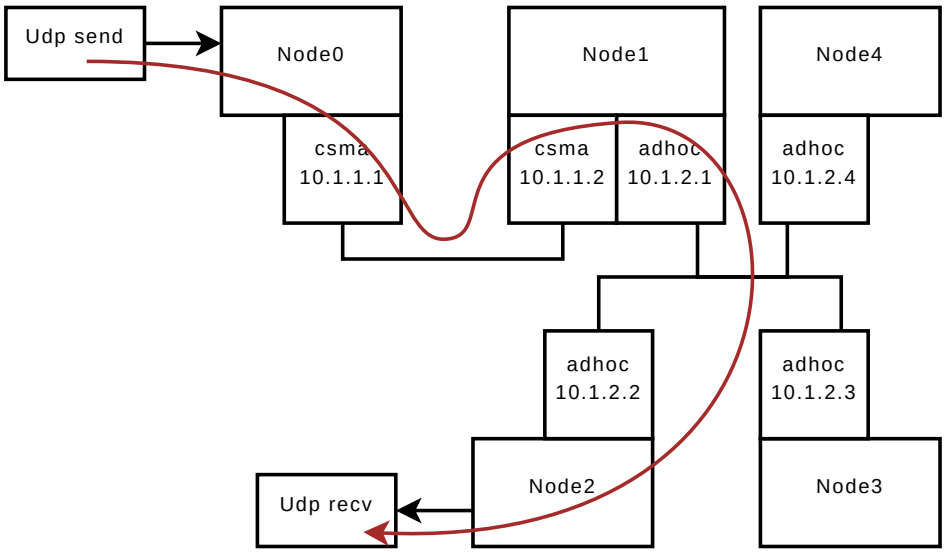
The ns-3 API

There are two ways to interact with the ns-3 API:

- Construct a simulation with the *Container* API:
  - Apply the same operations on sets of objects
  - Easy to build topologies with repeating patterns
- Construct a simulation with the *low-level* API:
  - Instanciate every object separately, set its attributes, connect it to other objects.
  - Very flexible but potentially complex to use

The best way to understand how they work and relate to each other is to use both on the same example

The Testcase



- One csma link
- One wifi infrastructure network
- Two ip subnetworks
- One udp traffic generator
- One udp traffic receiver
- Global god ip routing

The *Container* Version

Fire up an editor and look at the code

The *Low-Level* Version

Fire up an editor and look at the code

Why are objects so complicated to create ?

We do:

```
Ptr<Node> node0 = CreateObject<Node> ();
```

Why not:

```
Node *node0 = new Node ();
```

Or:

```
Node node0 = Node ();
```

Templates: the Nasty Brackets

- Contain a list of *type* arguments
- Parameterize a class or function from input type
- In ns-3, used for:
  - Standard Template Library
  - Syntactical sugar for low-level facilities
- Saves a lot of typing
- No portability/compiler support problem
- Sometimes painful to decipher error messages.

Memory Management

It is hard in C++:

- No garbage collector
- Easy to forget to delete an object
- Pointer cycles
- Ensure coherency and uniformity

So, we use:

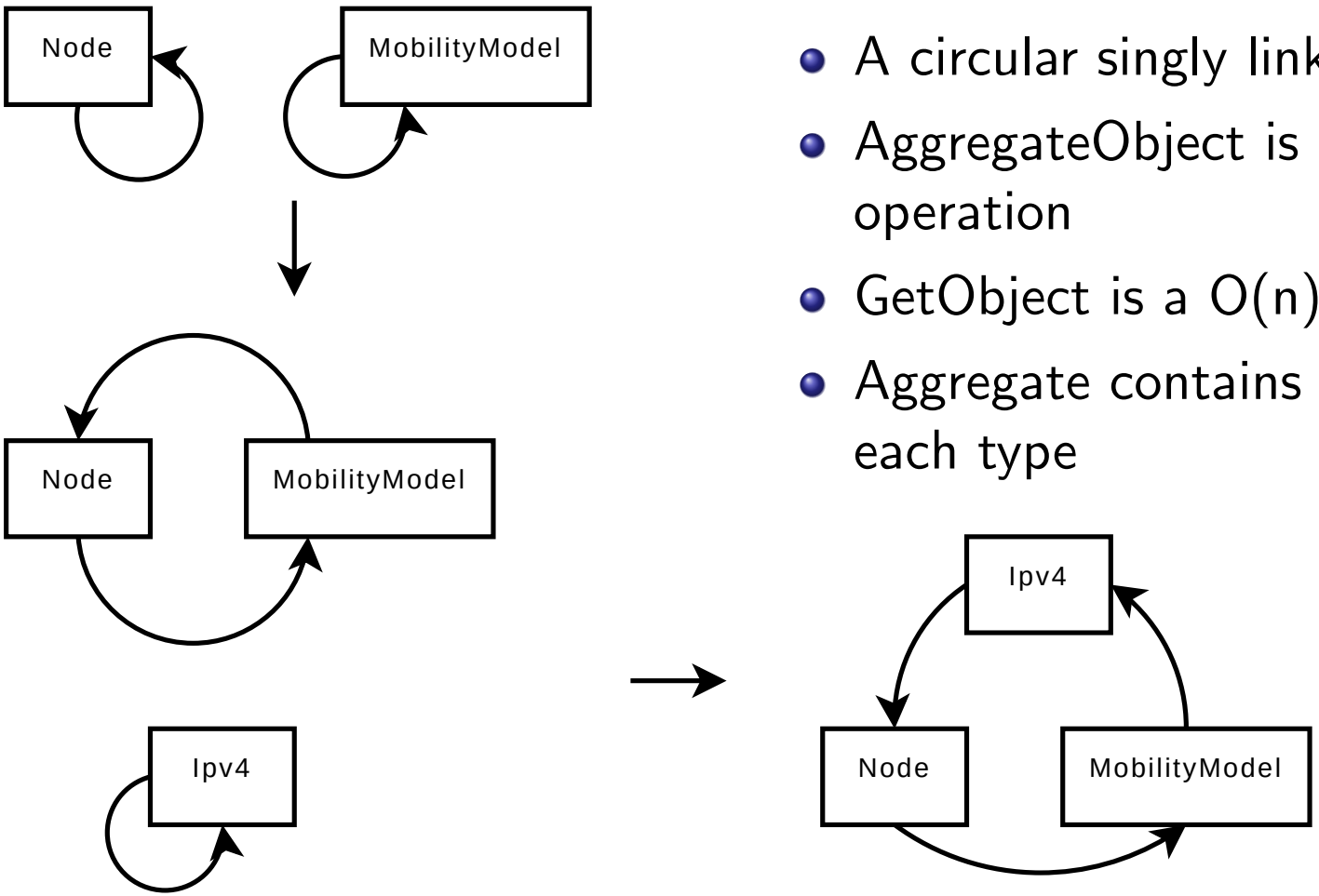
- Reference counting: track number of pointers to an object (`Ref+Unref`)
- Smart pointers: `Ptr<>`, `Create<>` and, `CreateObject<>`
- Sometimes, explicit `Dispose` to break cycles

Why don't we have a MobileNode ?

```
Ptr<Node> node = CreateObject<Node> ();
Ptr<MobilityModel> mobility = CreateObject<...> ();
node->AggregateObject (mobility);
```

- Some nodes need an IPv4 stack, a position, an energy model.
- Some nodes need just two out of three.
- Others need other unknown features.
- The obvious solution: add everything to the Node base class, but:
  - The class will grow uncontrollably over time
  - Everyone will need to patch the class
  - Slowly, every piece of code will depend on every other piece of code
  - A maintenance nightmare...
- A better solution:
  - Separate functionality belongs to separate classes
  - Objects can be aggregated at runtime to obtain extra functionality

Object aggregation



- A circular singly linked-list
- AggregateObject is a constant-time operation
- GetObject is a O(n) operation
- Aggregate contains only one object of each type

The ns-3 type system

- The aggregation mechanism needs information about the type of objects are runtime
- The attribute mechanism needs information about the attributes supported by a specific object
- The tracing mechanism needs information about the trace sources supported by a specific object

All this information is stored in `ns3::TypeId`:

- The parent type
- The name of the type
- The list of attributes (their name, their type, etc.)
- The list of trace sources (their name, their type, etc.)

The ns-3 type system

It is not very complicated to use:

- Derive from the `ns3::Object` base class
- Define a `GetTypeId` static method:

```
class Foo : public Object {
public:
  static TypeId GetTypeId (void);
};
```

- Define the features of your object:

```
  static TypeId tid = TypeId ("ns3::Foo")
    .SetParent<Object> ()
    .AddAttribute ("Name", "Help", ...)
    .AddTraceSource ("Name", "Help", ...);
  return tid;
```

- call `NS_OBJECT_ENSURE_REGISTERED`

Application Transmission I

User writes:

```
Ptr<Application> app = ...;
app->Start (Seconds (1.0));
```

`Application::Start`:

```
m_startEvent = Simulator::Schedule (startTime,
                &Application::StartApplication, this);
```

Application Transmission II

User calls `Simulator::Run`:

```
m_socket = Socket::CreateSocket (GetNode(), m_tid);
m_socket->Bind ();
m_socket->Connect (m_peer);
...
m_startStopEvent = Simulator::Schedule(offInterval,
                   &OnOffApplication::StartSending, this);
```

`Socket::CreateSocket`:

```
Ptr<SocketFactory> socketFactory;
socketFactory = node->GetObject<SocketFactory> (tid);
s = socketFactory->CreateSocket ();
```

Application Transmission III

`OnOffApplication::StartSending`:

```
m_sendEvent = Simulator::Schedule(nextTime,
                    &OnOffApplication::SendPacket, this);
```

`OnOffApplication::SendPacket`:

```
Ptr<Packet> packet = Create<Packet> (m_pktSize);
m_txTrace (packet);
m_socket->Send (packet);
...
m_sendEvent = Simulator::Schedule(nextTime,
                    &OnOffApplication::SendPacket, this);
```

Summary: how applications access network stacks

How to use a new protocol Foo:

```
Ptr<SocketFactory> factory =
   node->GetObject<FooSocketFactory> ();
Ptr<Socket> socket = factory->CreateSocket ();
socket->...
```

How to implement a new protocol Foo:

- Create FooSocketFactory, a subclass of SocketFactory
- Aggregate FooSocketFactory to a Node during topology construction (for UDP, done by InternetStackHelper::Install)
- From FooSocketFactory::CreateSocket, create instances of type FooSocket, a subclass of Socket

Note: Magic COW Packets I

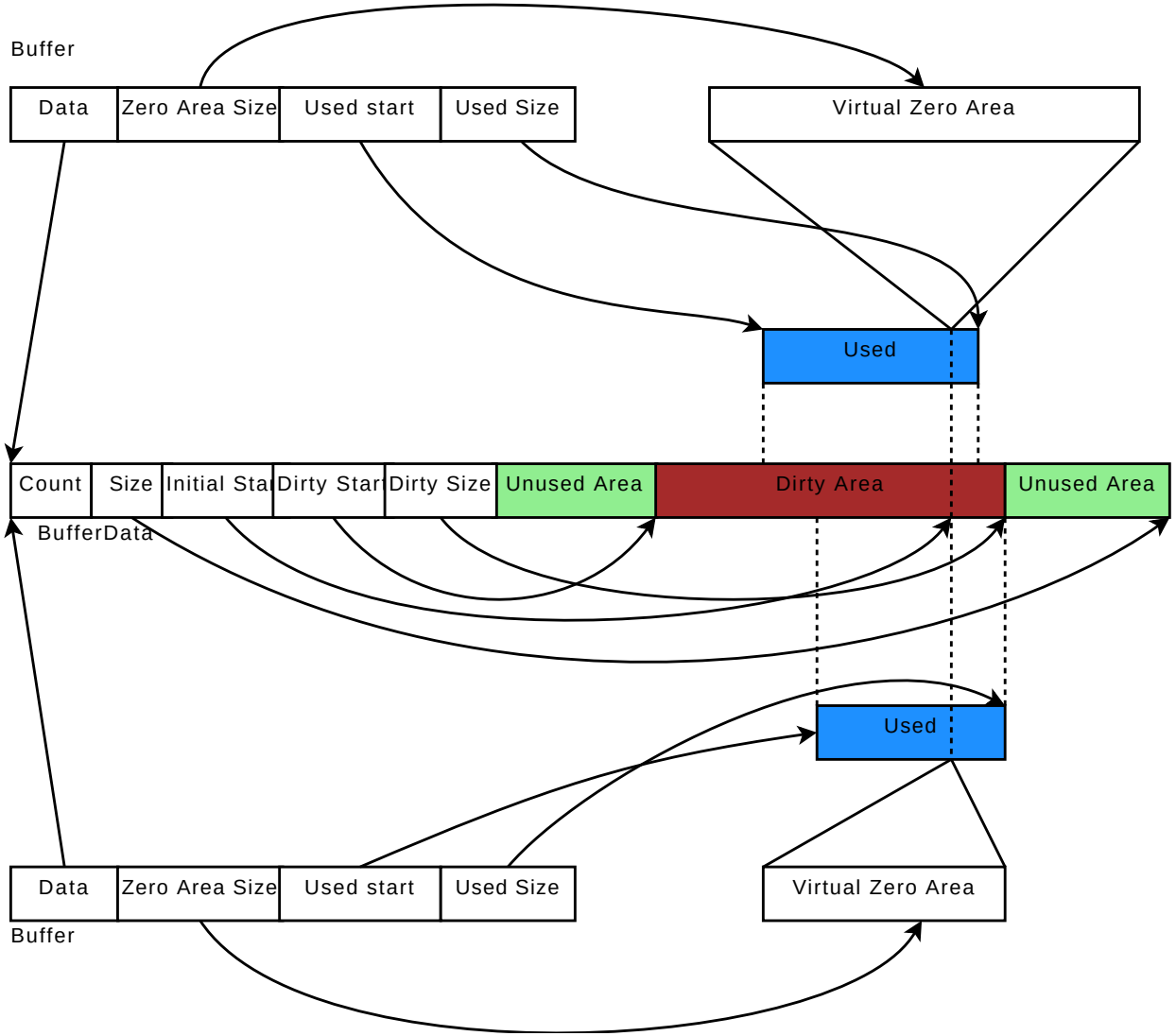
ns-3 packets contain a lot of information:

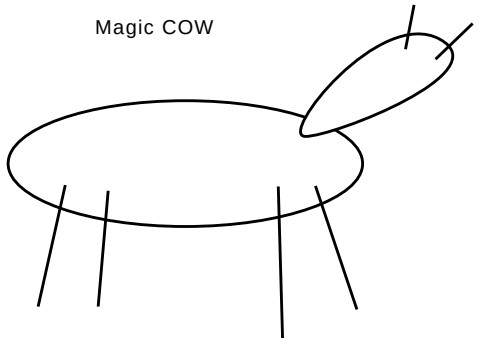
- Buffer: a byte buffer which contains payload, headers, trailers, all in real network format
- Metadata: information about the type of headers and trailers located in the byte buffer
- Tags: extra user-provided information, very useful for end-to-end simulation-only stuff: timestamps for rtt calculations, etc.

ns-3 packets are magic:

- They are reference-counted
- They have Copy On Write semantics: `Packet::Copy` does not create a new packet buffer: it creates a new reference to the same packet buffer
- Payload is zero-filled and never allocated by default: only headers and trailers use memory

Note: Magic COW Packets II





UDP Transmission I

`UdpSocketImpl::Send` eventually calls `UdpSocketImpl::DoSendTo` which calls `UdpL4Protocol::Send`:

```
UdpHeader udpHeader;
...
udpHeader.SetDestinationPort (dport);
udpHeader.SetSourcePort (sport);
packet->AddHeader (udpHeader);
Ptr<Ipv4L3Protocol> ipv4 =
        m_node->GetObject<Ipv4L3Protocol> ();
ipv4->Send (packet, saddr, daddr, PROT_NUMBER);
```

IPv4 Transmission I

Ipv4L3Protocol::Send:

```
Ipv4Header ipHeader;
...
ipHeader.SetSource (source);
ipHeader.SetDestination (destination);
ipHeader.SetProtocol (protocol);
ipHeader.SetPayloadSize (packet->GetSize ());
...
ipHeader.SetTtl (...);
...
Lookup (ipHeader, packet,
   MakeCallback (&Ipv4L3Protocol::SendRealOut, this));
```

IPv4 Transmission II

`Ipv4L3Protocol::Lookup` searches a protocol which has an outgoing route for the packet and calls `Ipv4L3Protocol::SendRealOut`:

```
packet->AddHeader (ipHeader);
Ptr<Ipv4Interface> outInterface =
   GetInterface (route.GetInterface ());
outInterface->Send (packet, ipHeader.GetDestination ())
```

Down, in `ArpIpv4Interface`:

```
Ptr<ArpL3Protocol> arp = m_node->GetObject<ArpL3Protocol> ();
Address hardwareDestination;
arp->Lookup (p, dest, GetDevice (), m_cache, &hardwareDestinat
GetDevice ()->Send (p, hardwareDestination,
                    Ipv4L3Protocol::PROT_NUMBER);
```

Note: How Do you Implement a new Header ? I

The class declaration:

```
class MyHeader : public Header
...
  void SetData (uint16_t data);
  uint16_t GetData (void) const;
...
  static TypeId GetTypeId (void);
  virtual TypeId GetInstanceTypeId (void) const;
  virtual void Print (std::ostream &os) const;
  virtual void Serialize (Buffer::Iterator start) const;
  virtual uint32_t Deserialize (Buffer::Iterator start);
  virtual uint32_t GetSerializedSize (void) const;
private:
  uint16_t m_data;
```

Note: How Do you Implement a new Header ? II

The implementation:

```
void
MyHeader::Serialize (Buffer::Iterator start) const
{
  start.WriteHtonU16 (m_data);
}
uint32_t
MyHeader::Deserialize (Buffer::Iterator start)
{
  m_data = start.ReadNtohU16 ();
  return 2;
}
```

Note: How Do you Implement a new Header ? III

What really matters:

- Copy/Paste the code for `GetTypeId` and `GetInstanceTypeId`
- Make sure `GetSerializedSize` returns enough for `Serialize`
- Make sure `Hton` are balanced with `Ntoh`
- Remember that what is written in `Buffer::Iterator` must be faithful the the real network representation of the protocol header

Arp

`ArpL3Protocol::Lookup`:

- Try to find a matching live entry
- If needed, send an ARP request on `NetDevice::Send`
- Wait for reply

`ArpL3Protocol::Receive`:

- If request for us, send reply
- If reply, check if request pending, update cache entry, flush packets from cache entry

CsmaNetDevice Transmission

`CsmaNetDevice::Send`:

- Add ethernet header and trailer
- Queue packet in tx queue
- Perform backoff if medium is busy
- When medium is idle, start transmission (delay is bytes*8/throughput)
- When transmission completes, request packet forwarding on medium

`CsmaChannel::TransmitEnd`:

- Apply propagation delay on transmission
- Distribute packet to all devices on the medium for reception

`CsmaNetDevice::Receive`:

- Remove ethernet header and trailer
- Filter unwanted packets
- Apply packet error model
- Call device *receive* callback

Summary: From layer 2 to layer 3

During topology setup:

- Call `Node::RegisterProtocolHandler` to register a layer 3 protocol handler by its protocol number
- `Node::AddDevice` sets device *receive* callback to `Node::NonPromiscReceiveFromDevice`

At runtime:

- Device calls *receive* callback to send packet to layer 3
- `Node::NonPromiscReceiveFromDevice` searches matching protocol handlers by protocol number

IPv4 Reception

`Ipv4L3Protocol::Receive`:

- Remove IPv4 header, verify checksum
- Forward packet to matching raw IPv4 sockets
- If needed, forward packet down to outgoing interfaces
- If needed, forward packet up the stack to matching layer 4 protocol with `Ipv4L3Protocol::GetProtocol`

Wifi Transmission

`WifiNetDevice::Send` is fairly simple:

```
LlcSnapHeader llc;
llc.SetType (protocolNumber);
packet->AddHeader (llc);
m_txLogger (packet, realTo);
m_mac->Enqueue (packet, realTo);
```

It's an AP so, in `NqapWifiMac::ForwardDown`:

```
  WifiMacHeader hdr;
  hdr.SetAddr1 (to);
  hdr.SetAddr2 (GetAddress ());
  hdr.SetAddr3 (from);
...
  m_dca->Queue (packet, hdr);
```

Wifi Transmission: DcaTxop

DcaTxop::Queue:

- Queue outgoing packet in WifiMacQueue
- Use DCF (DcfManager and DcfState to obtain a tx opportunity

When the tx opportunity happens, DcaTxop::NotifyAccessGranted is called:

- Dequeue packet
- Prepare the first fragment if needed
- Enable RTS if needed
- Call MacLow::StartTransmission
- Wait for notifications about transmission success or failure from MacLow
- Eventually, start retransmissions, send more fragments

Wifi Transmission: MacLow I

- MacLow::StartTransmission starts a CtsTimeout or an AckTimeout timer and, then calls WifiPhy::SendPacket:

```
if (m_txParams.MustSendRts ())
  SendRtsForPacket ();
else
  SendDataPacket ();
```

- MacLow::CtsTimeout and MacLow::NormalAckTimeout notify upper layers

Wifi PHY: layer 1

From the perspective of layer 2, it is a black box whose content is the topic of some presentations this afternoon !

Wifi Reception: MacLow

MacLow::ReceiveOk handles incoming packets:

```
WifiMacHeader hdr;
packet->RemoveHeader (hdr);
if (hdr.IsRts ())
  ...
else if (hdr.IsCts () &&
  ...
else if (hdr.IsAck () &&
  ...
else if (hdr.GetAddr1 () == m_self)
  ...
else if (hdr.GetAddr1 ().IsGroup ())
  ...
```

And notifies upper layers with its *receive* callback

Wifi Reception: Defragmentation, Duplicate Detection

MacRxMiddle::Receive:

```
if (IsDuplicate (hdr, originator))
  return;
Ptr<Packet> agregate = HandleFragments (packet, hdr, originato
if (agregate == 0)
  return;
m_callback (agregate, hdr);
```

Wifi Reception: MacHigh

- NqstaWifiMac::Receive:

```
else if (hdr->IsData ())
  ...
else if (hdr->IsProbeReq () ||
         hdr->IsAssocReq ())
  ...
else if (hdr->IsBeacon ())
  ...
else if (hdr->IsProbeResp ())
  ...
else if (hdr->IsAssocResp ())
  ...
```

- WifiNetDevice::ForwardUp: call the device *receive* callback

Summary: From Layer 3 to Layer 4

- During topology setup, call `Ipv4L3Protocol::Insert` to register a layer 4 protocol with its protocol number
- At runtime:
  - Call `Ipv4L3Protocol::GetProtocol`
  - Call `Ipv4L4Protocol::Receive`

UDP Reception

```
UdpHeader udpHeader;
packet->RemoveHeader (udpHeader);
Ipv4EndPointDemux::EndPoints endPoints =
  m_endPoints->Lookup (destination,
                       udpHeader.GetDestinationPort (),
                       source,
                       udpHeader.GetSourcePort (), ...);
for (endPoint = endPoints.begin ();
     endPoint != endPoints.end (); endPoint++)
  {
    (*endPoint)->ForwardUp (...);
  }
```

Ipv4EndPoint::ForwardUp calls into UdpSocketImpl::ForwardUp

Application Reception I

`UdpSocketImpl::ForwardUp`

```
if ((m_rxAvailable + packet->GetSize ()) <= m_rcvBufSize)  {
  m_deliveryQueue.push (packet);
  m_rxAvailable += packet->GetSize ();
  NotifyDataRecv ();
}
```

`PacketSink::HandleRead`:

```
packet = socket->RecvFrom (from)
```

Application Reception II

UdpSocketImpl::Recv

```
if (m_deliveryQueue.empty() )
  {
    m_errno = ERROR_AGAIN;
    return 0;
  }
Ptr<Packet> p = m_deliveryQueue.front ();
if (p->GetSize () <= maxSize)
  {
    m_deliveryQueue.pop ();
    m_rxAvailable -= p->GetSize ();
  }
return p;
```

Pause !!

Let's have a break

Part III

Experimentation

Outline

What is experimentation ?

- Physical testbeds:
  - Cluster of ethernet networked machines
  - Set of laptops with wireless cards, on a desktop
  - Set of wireless motes in backpacks
  - Planetlab, ORBIT, Onelab, Emulab, ...
- Simulation:
  - ns-2, ns-3, nctuns, omnetpp etc.
- Emulation:
  - Coupling of simulation with physical testbeds or real world

The need for experimentation

You have designed a brand new algorithm (say, TCP Reno):

- Must explore parameter space
- Must verify that, once deployed, it won't kill your network,
- Must perform a couple of performance measures
- Must verify that it works !

What is hard about experimentation

- Reproducibility: scientific methodology means that you must publish reproducible results
- Configuration: large scale experiments require a lot of configuration
- Instrumentation: need to gather data about the behavior of the experiment to figure out what happened
- Fidelity:did the experiment really capture the effects you are really interested in ?

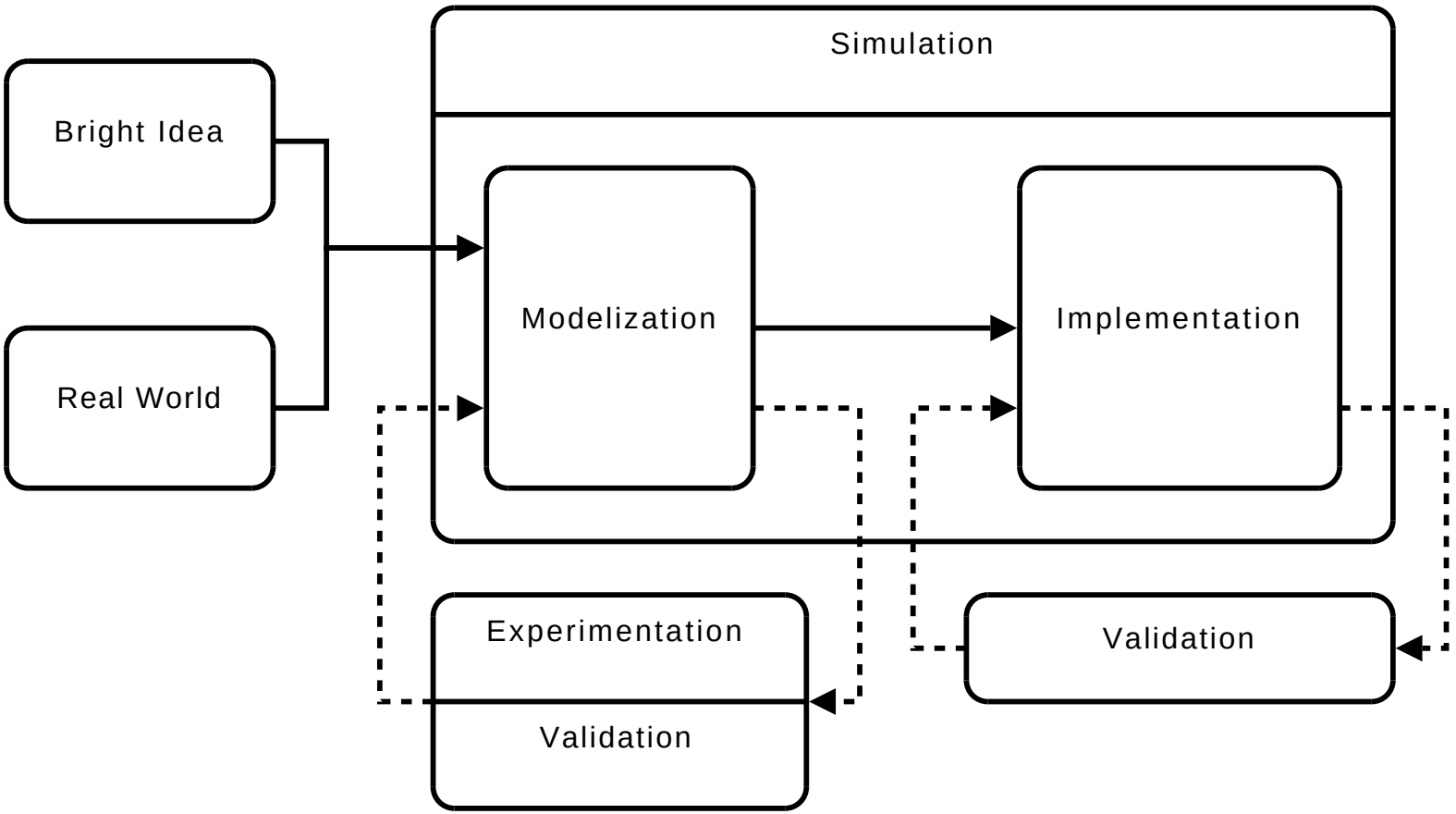
Simulation vs testbeds

- Simulation is:
  - Reproducible,
  - Easy to configure,
  - Easy to instrument,
  - Potentially unrealistic
- Testbeds are:
  - Potentially more realistic than simulation,
  - More or less reproducible (often less),
  - Hard to configure,
  - Very hard to instrument

The ideal workflow

1. A great idea,
2. Thorough simulation,
3. Thorough testbed experimentation,
4. Small-scale deployment,
5. Large-scale deployment

The ideal workflow



The key issue: cost of experimentation

- Simulation:
  - cost of model validation
  - cost of model implementation
- Cost of transition from simulation to testbed experimentation
- Cost of transition from testbed experimentation to deployment

Ns-3 goals: minimizing impedence mismatch

ns-3 model architecture close to real-world:

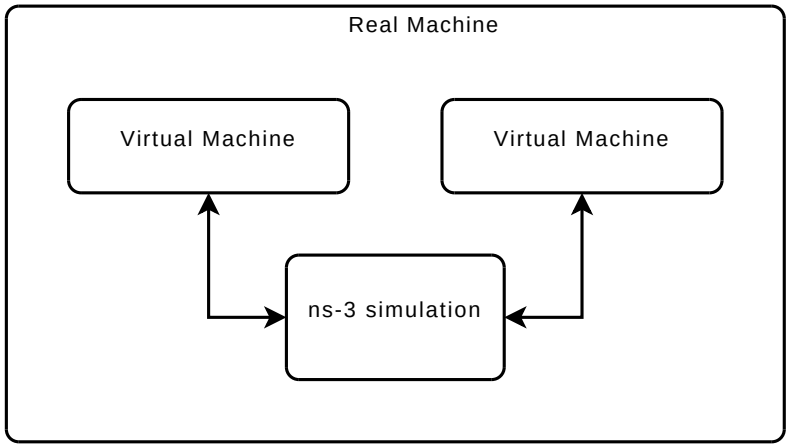
- multi-interface nodes
- real ip addressing
- socket API
- device API
- packets use on-the-wire buffer format
- natively support emulation

Write once, run everywhere

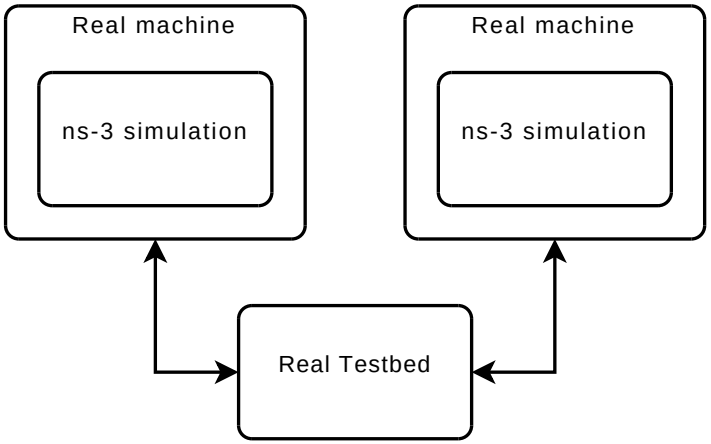
- POSIX socket application:
  - can be run in simulation (ns-3-simu)
  - can be run in testbed
  - can be deployed
- Kernel network stack:
  - can be run in simulation (ns-3+nsc)
  - can be run in testbed
  - can be deployed
- Emulation:
  - can be used to integrate simulator in testbed
  - mixed experiments (simulation+testbed)

Ns-3 Emulation modes

Ns-3 interconnects real or virtual machines:



Testbeds interconnect ns-3 stacks



Various combinations and hybrids of the above are possible

Example: CORE and ns-3

A Scalable Network Emulator:

- Network lab "in a box":
  - Efficient and scalable
  - Easy-to-use GUI canvas
- Kernel-level networking efficiency:
  - Reference passing packet sending
  - Kernel-level packet delays
- Runs real binary code: no need to modify applications
- Connects with real networks
  - Hardware-in-the-loop
  - Distributed - runs on multiple servers
  - Virtual nodes process real packets
- Fork of the IMUNES project: University of Zagreb
- Open Source: `http://cs.itd.nrl.navy.mil/work/core`

Binary loader

The goal is to be able to load the same binary multiple times in the same simulation process:

- Uses a special ELF loader
- Implements posix socket API using ns-3 sockets
- Can debug all applications with a single debugger

Part IV

Practical exercise

The helper example

- Drop `tutorial-helper.cc` in `scratch` directory
- Run `./waf`
- Run `./waf --shell`
- Run `./build/debug/scratch/tutorial-helper`

Emulation

- Look at `examples/tap-wifi-dumbbell.cc`
- Build it, become root
- Run `./waf --shell`
- Run `./build/debug/examples/tap-wifi-dumbbell`
- In another terminal, `ping 10.1.1.1`, `ping 10.1.1.3`, `ping 10.1.3.1`
- Add a route to `10.1.3.0`:

```
route add -net 10.1.3.0 netmask 255.255.255.0 dev thetap gw 10.1.1.2
```

- `ping 10.1.3.1`

Exploring the system I

- Attempt to build an optimized version of this (`./waf configure -d optimized`)
- Look at the ascii output of the script (`tutorial-helper.tr`)
- Look at the pcap output with wireshark, tcpdump (`tutorial-helper-nodeid-deviceid.pcap`)
- Notice the ARP packets
- Notice the IP packets with invalid IP checksum
- Try `--help`, `--ns3::Ipv4L3Protocol::CalcChecksum`, and `--ns3::UdpL4Protocol::CalcChecksum`

Exploring the system II

- Replace `GtkConfigStore` with `ConfigStore`, use `--PrintAttributes=ns3::ConfigStore`
- Try `--ns3::ConfigStore::Mode=Save`, `--ns3::ConfigStore::Filename=config.xml` and `--ns3::ConfigStore::FileFormat=Xml`
- Look at what is going on in the wifi layer: `NS_LOG=MacLow`
- Try `NS_LOG=Ipv4L3Protocol`
- Try `--RngRun=2`, `NS_GLOBAL_VALUE=RngRun=2`

Let's try to use TCP

- The native ns-3 TCP: use `ns3::TcpSocketFactory` in the script
- Use `GtkConfigStore` to set `ns3::TcpSocketFactory`
- Introduce a second UDP generator from node 0 to 1 at second 2.0: observe qualitatively TCP adaptation

Tracing TCP behavior I

Declare a trace sink:

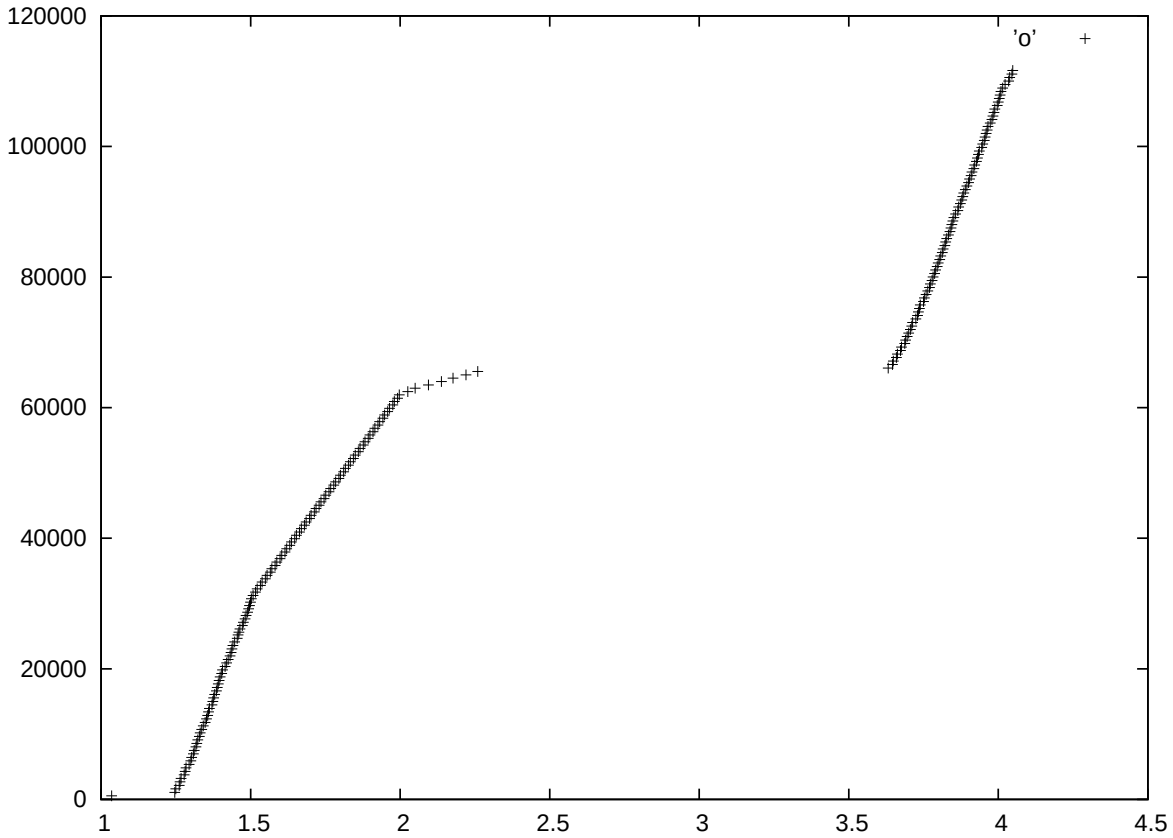
```
static uint32_t g_totalBytes = 0;
static void
TcpRx (std::string ctx, Ptr<const Packet> packet,
       const Address &)
{
  g_totalBytes += packet->GetSize ();
  std::cout << Simulator::Now ().GetSeconds ()
            << " " << g_totalBytes << std::endl;
}
```

Connect the trace sink to a trace source:

```
Config::MatchContainer match;
match = Config::LookupMatches ("/NodeList/2/ApplicationList/0");
match.Connect ("Rx", MakeCallback (&TcpRx));
```

Tracing TCP behavior II

Use spreadsheet/gnuplot to view the trace:



Part V

We are done, finally

In the future

- Load normal posix socket applications in simulator (quagga, xorp, bittorrent, etc.)
- Graphically build simple and mixed simulation/experiment topologies: CORE, NEF
- Ipv6 integration
- Wifi: More PHY models, 802.11e,n,s MAC implementations
- Parallel simulation support for large-scale simulations using MPI on clusters
- Parallel simulation support for fast simulations on multicore machines

Resources

- Web site: `http://www.nsnam.org`
- Mailing list: `http://mailman.isi.edu/mailman/listinfo/ns-developers`
- IRC: #ns-3 at freenode.net
- Tutorial: `http://www.nsnam.org/docs/tutorial/tutorial.html`
- Code server: `http://code.nsnam.org`
- Wiki: `http://www.nsnam.org/wiki/index.php/Main_Page`

Acknowledgments

All slides stolen from other's presentations and tutorials (in no particular order):

- Tom Henderson
- Gustavo Carneiro
- Joseph Kopena
- Sam Jansen

Part VI

Extras

Callback objects I

Ns-3 Callback class implements function objects

- Type safe callbacks, manipulated by value
- Used for example in sockets and tracing

Wrap a raw function:

```
double MyFunc (int x, float y) {
  return double (x + y) / 2;
}
Callback<double, int, float> cb1;
cb1 = MakeCallback (MyFunc);
double result = cb1 (2,3);
```

Callback objects II

Wrap a member method:

```
Class MyClass {
public:
  double MyMethod (int x, float y) {
     return double (x + y) / 2;
  };
};
Callback<double, int, float> cb1;
MyClass myobj;
cb1 = MakeCallback(&MyClass::MyMethod, &myobj);
double result = cb1 (2,3);
```

Callback objects III

Wrap a raw function and bind its first argument:

```
double MyFunc (int x, float y) {
  return double (x + y) / 2;
}
Callback<double, float> cb1;
cb1 = MakeBoundCallback (MyFunc, 2);
double result = cb1 (3);
```

Debugging support I

- Assertions: `NS_ASSERT (expression);`
  - Aborts the program if expression evaluates to false
  - Includes source file name and line number
  - Compiled in only in debug builds
- Unconditional Breakpoints: `NS_BREAKPOINT ();`
  - Forces an unconditional breakpoint, always 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_DEBUG ("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 II

- 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`