# ns-3 Training

**Session 2: Thursday**

**MNM Workshop**
**March 2015**

# ns-3 build systems

# Software introduction

- Download the latest release
  - `wget http://www.nsnam.org/releases/ns-allinone-3.19.tar.bz2`
  - `tar xjf ns-allinone-3.19.tar.bz2`
- Clone the latest development code
  - `hg clone http://code.nsnam.org/ns-3-allinone`

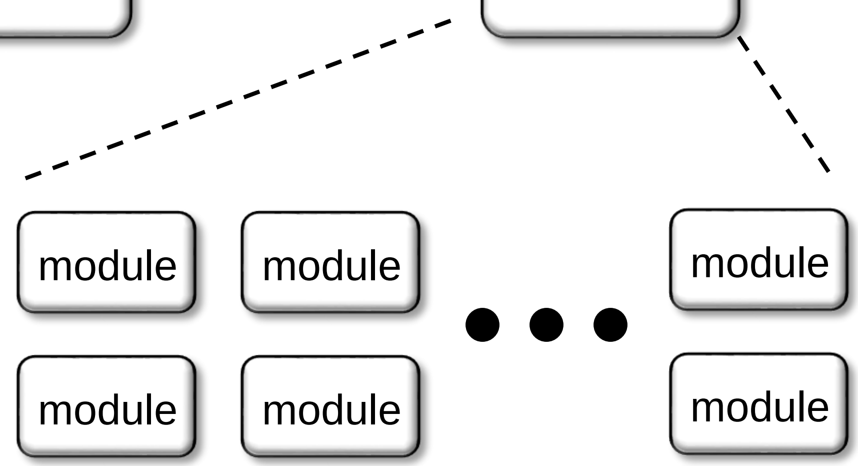
Q. What is "**hg clone**"?
A. Mercurial (http://www.selenic.com) is our source code control tool.

# Software building

- Two levels of ns-3 build

1) **bake** (a Python-based build system to control an ordered build of ns-3 and its libraries)


2) **waf**, a build system written in Python




3) **build.py** (a custom Python build script to control an ordered build of ns-3 and its libraries) **<--- may eventually be deprecated**

# ns-3 uses the 'waf' build system

- Waf is a Python-based framework for configuring, compiling and installing applications.
  - It is a replacement for other tools such as Autotools, Scons, CMake or Ant
  - http://code.google.com/p/waf/
- For those familiar with autotools:
- `configure` → `./waf configure`
- `make` → `./waf build`

# waf configuration

- Key waf configuration examples

```
./waf configure
    --enable-examples
    --enable-tests
    --disable-python
    --enable-modules
```

- Whenever build scripts change, need to reconfigure

Demo: `./waf --help`

`./waf configure --enable-examples --enable-tests --enable-modules='core'`

Look at: `build/c4che/_cache.py`

# wscript example

```
## -*- Mode: python; py-indent-offset: 4; indent-tabs-mode: nil; coding: utf-8; -*-

def build(bld):
    obj = bld.create_ns3_module('csma', ['network', 'applications'])
    obj.source = [
        'model/backoff.cc',
        'model/csma-net-device.cc',
        'model/csma-channel.cc',
        'helper/csma-helper.cc',
        ]
    headers = bld.new_task_gen(features=['ns3header'])
    headers.module = 'csma'
    headers.source = [
        'model/backoff.h',
        'model/csma-net-device.h',
        'model/csma-channel.h',
        'helper/csma-helper.h',
        ]

    if bld.env['ENABLE_EXAMPLES']:
        bld.add_subdirs('examples')

    bld.ns3_python_bindings()
```

# waf build

- Once project is configured, can build via `./waf build` or `./waf`
- waf will build in parallel on multiple cores
- waf displays modules built at end of build

Demo: `./waf build`

Look at: `build/` libraries and executables

# Running programs

- `./waf shell` provides a special shell for running programs
  - Sets key environment variables

```
./waf --run sample-simulator
./waf --pyrun src/core/examples/sample-
  simulator.py
```

# Build variations

- Configure a build type is done at waf configuration time
- debug build (default): all asserts and debugging code enabled

```
./waf -d debug configure
```

- optimized

```
./waf -d optimized configure
```

- static libraries

```
./waf --enable-static configure
```

# Controlling the modular build

- One way to disable modules:
  - `./waf configure --enable-modules='a','b','c'`
- The `.ns3rc` file (found in utils/ directory) can be used to control the modules built
- Precedence in controlling build
  1) command line arguments
  2) .ns3rc in ns-3 top level directory
  3) .ns3rc in user's home directory

Demo how .ns3rc works

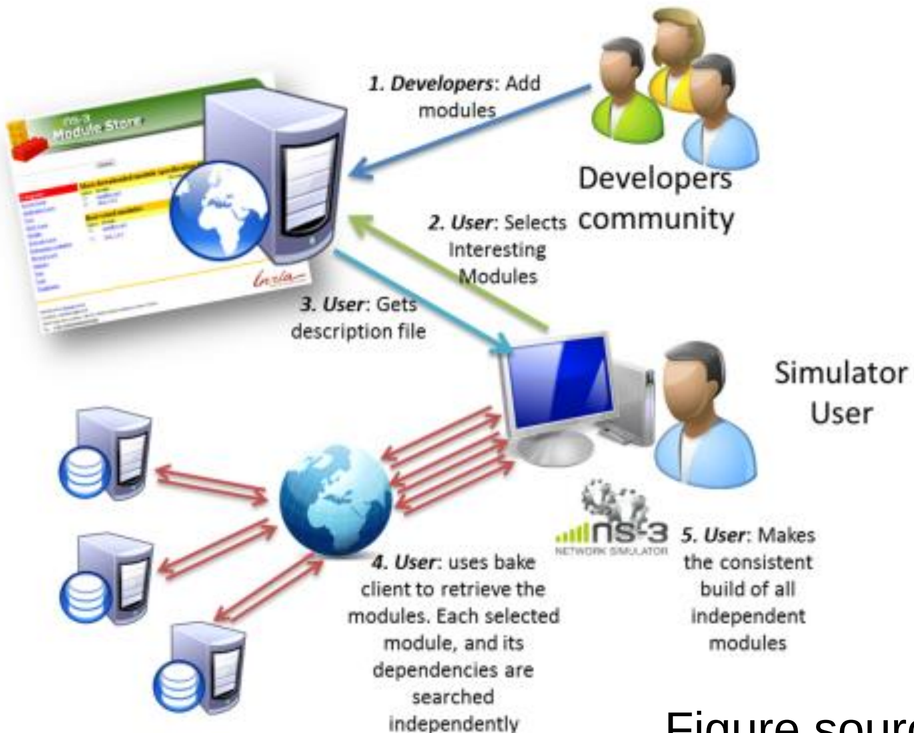
# Building without wscript

- The scratch/ directory can be used to build programs without wscripts

Demo how programs can be built without wscripts

# bake overview

- Open source project maintains a (more stable) core
- Models migrate to a more federated development process




"bake" tool (Lacage and Camara)

Components:

- build client
- "module store" server
- module metadata

Figure source: Daniel Camara

# bake basics

- bake can be used to build the Python bindings toolchain, Direct Code Execution, Network Simulation Cradle, etc.
- Manual available at https://www.nsnam.org/docs/bake/tutorial/html/index.html

```
./bake.py configure -e <module>
./bake.py show
./bake.py download
./bake.py build
```

# Placeholder slide for demoing bake

Demo: `./waf build`

Look at: `build/` libraries and executables

# Simulator core

# 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;
```

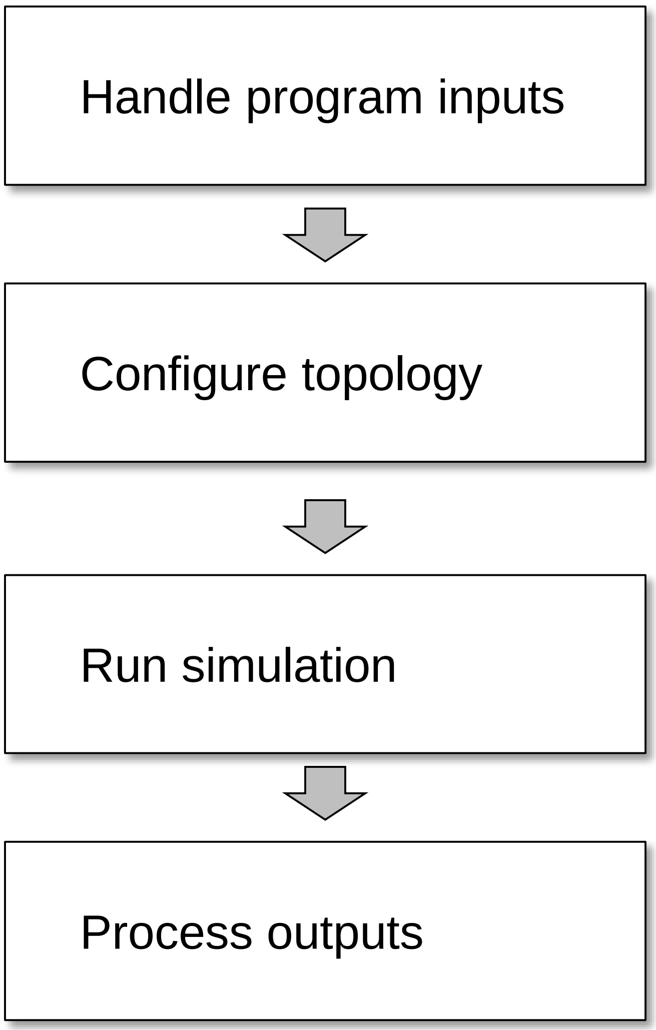
```
int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);

  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



# 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

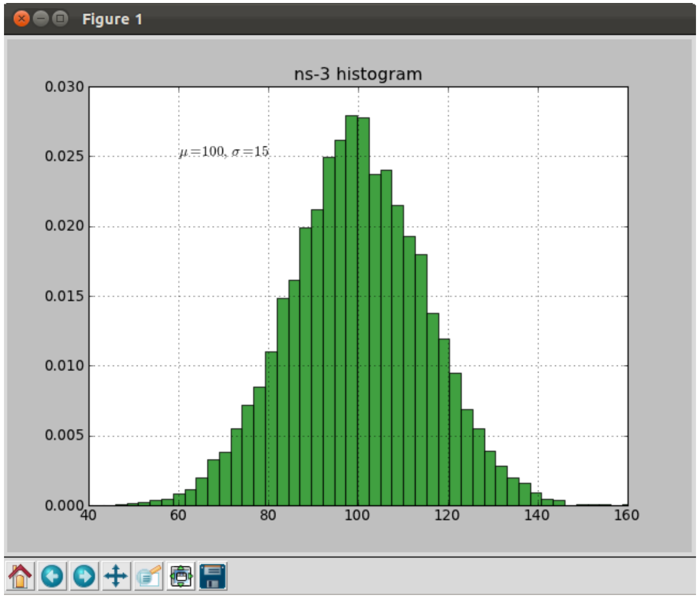
- The Simulator class holds a scheduler, and provides the API to schedule events, start, stop, and cleanup memory
- Several scheduler data structures (calendar, heap, list, map) are possible
- "RealTime" simulation implementation aligns the simulation time to wall-clock time
  - two policies (hard and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

# ns-3 Objects

NETWORK SIMULATOR

# 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

ns-3 NETWORK SIMULATOR

# 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

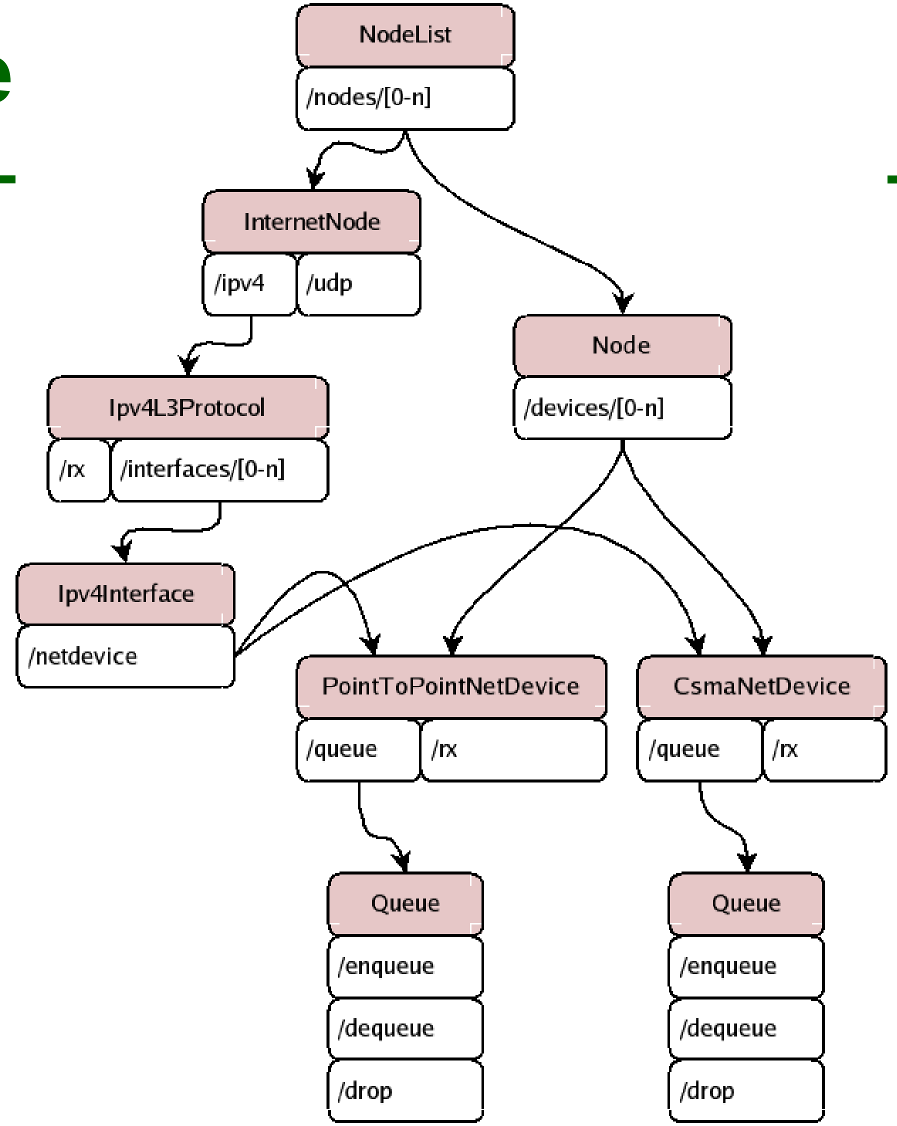
- The traditional C++ way:
  - export attributes as part of a class's public API
  - walk pointer chains (and iterators, when needed) to find what you need
  - use static variables for defaults
- The attribute system provides a more convenient API to the user to do these things

# 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"));

ns-3 NETWORK SIMULATOR

# 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/$ns3::WifiNetDevice/Phy/$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

ns-3 NETWORK SIMULATOR

# Attribute documentation

Main Page | Related Pages | Modules | Namespaces | Classes | Files

## 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"));
```

ns-3 NETWORK SIMULATOR

# 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

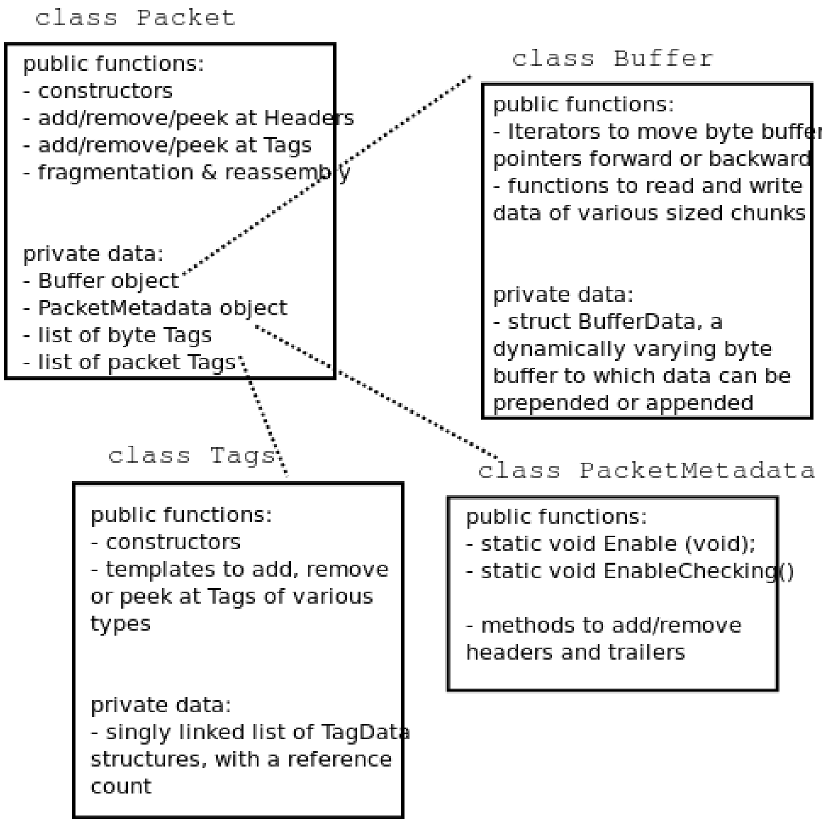
# Packets

# 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



# Copy-on-write

- Copy data bytes only as needed

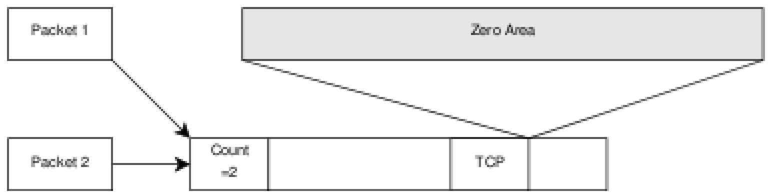



Figure 3.8: The TCP and the IP stacks hold references to a shared buffer.

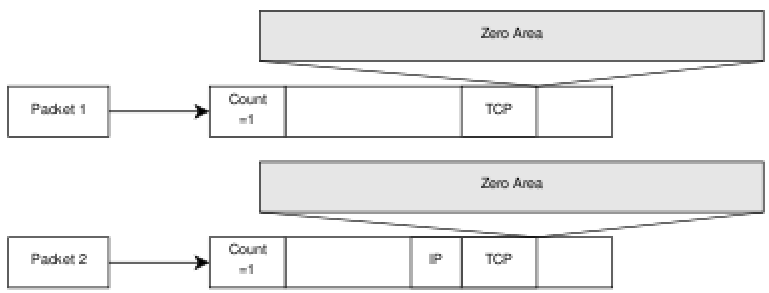



Figure 3.9: The IP stack inserts the IP header, triggers an un-share operation, completes the insertion.

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);
```

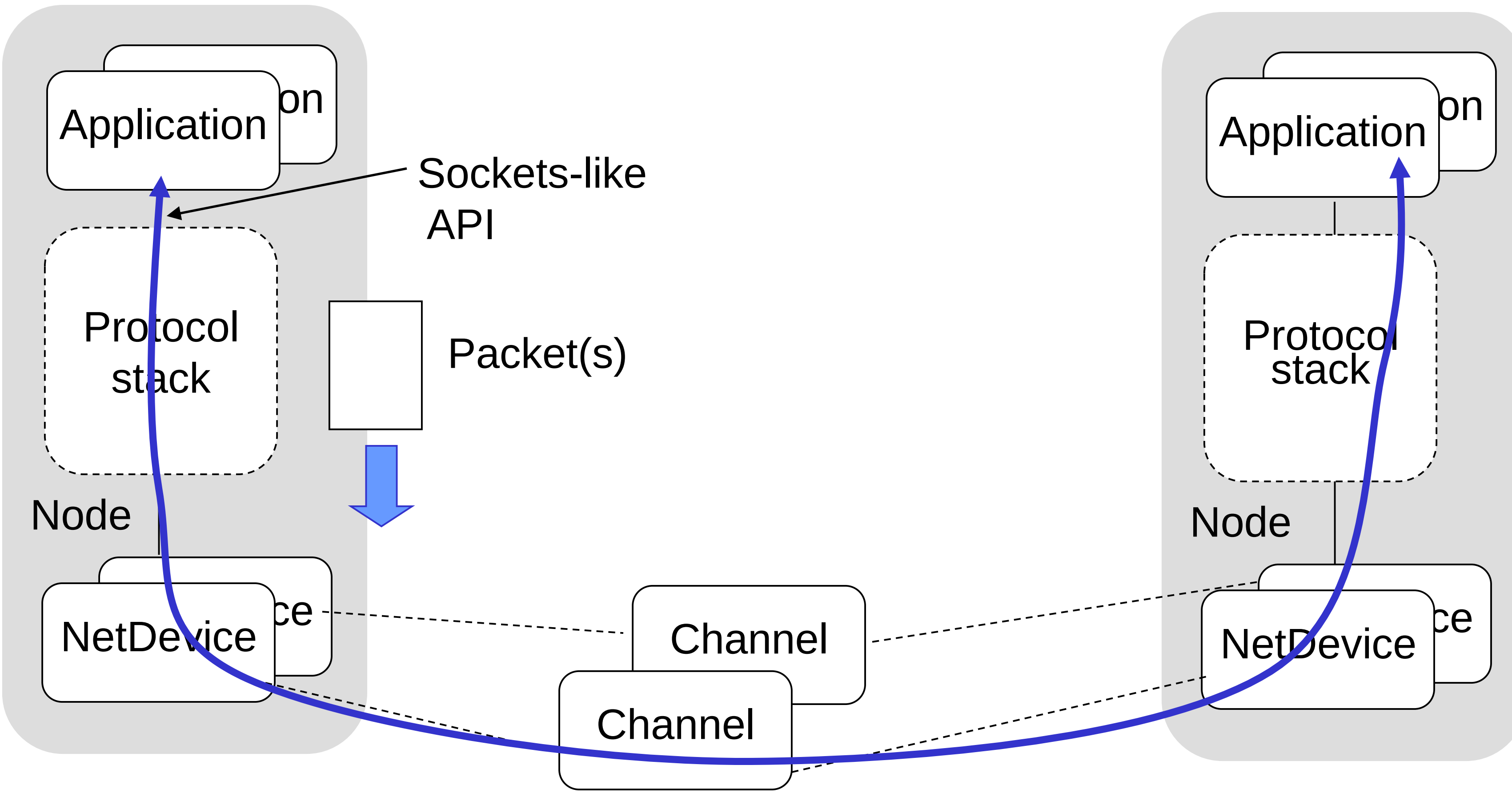
# Packet tags

- 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

# Walkthrough of M/M/1 queue

## Nodes and Devices

# The basic model



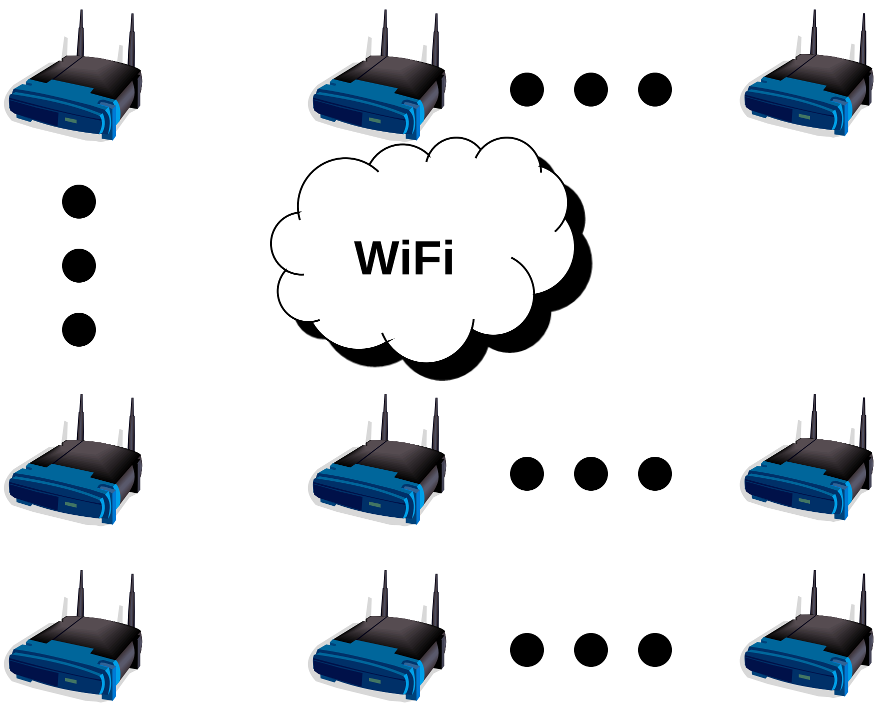
# Example program

- `examples/wireless/wifi-simple-adhoc-grid.cc`
- examine wscript for necessary modules
  - `'internet', 'mobility', 'wifi', 'config-store', 'tools'`
  - we'll add `'visualizer'`
- `./waf configure --enable-examples --enable-modules=...`

# Example program

- (5x5) grid of WiFi ad hoc nodes
- OLSR packet routing
- Try to send packet from one node to another

Source (node 24) by default




- Goal is to read and understand the high-level ns-3 API

Sink (node 0) by default

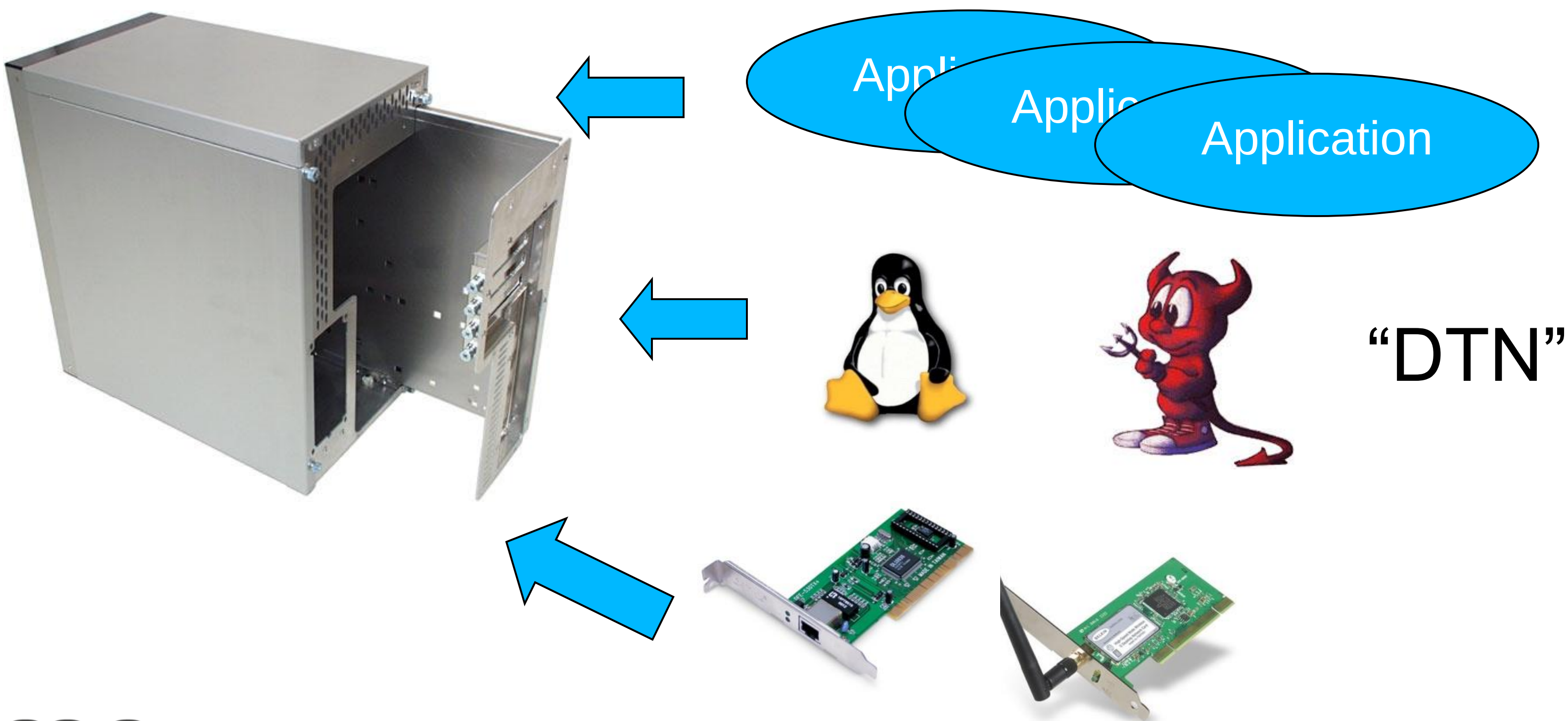
# Fundamentals

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

Nodes contain Applications, “stacks”, and NetDevices

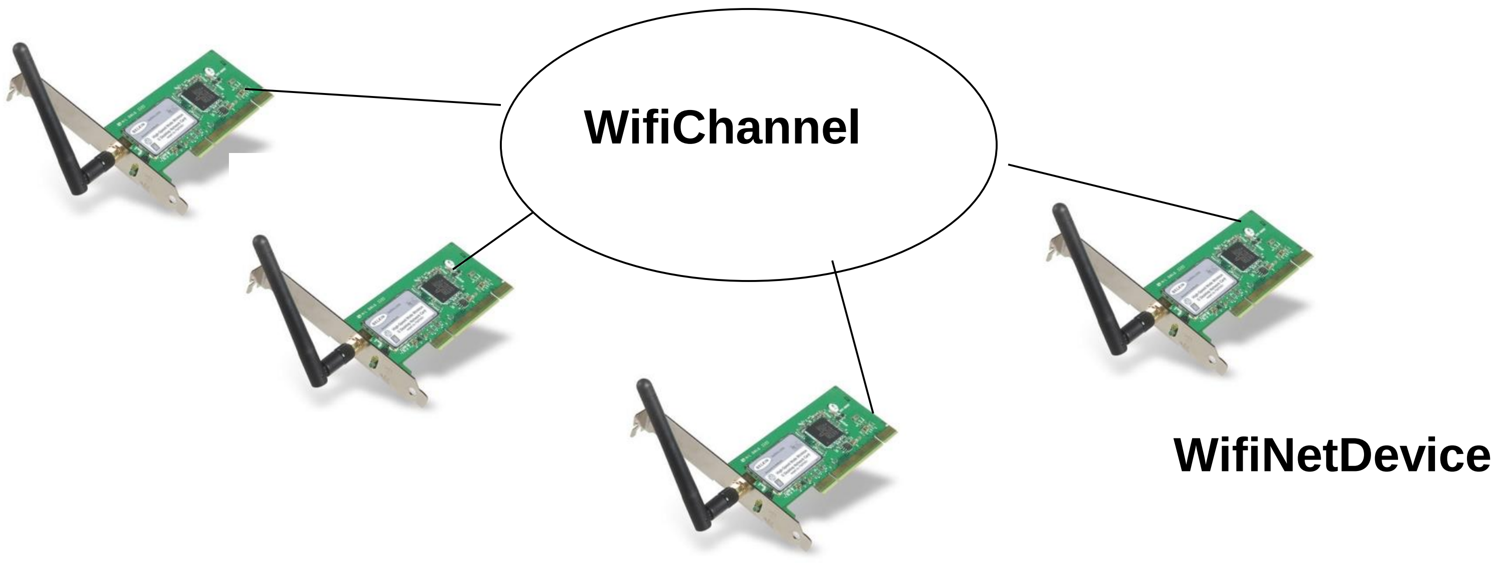
# Node basics

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




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
}
```

# Review of example program

```
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);

YansWifiChannelHelper wifiChannel;
wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel");
wifiPhy.SetChannel (wifiChannel.Create ());

// Add a non-QoS upper mac, and disable rate control
NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifi.SetStandard (WIFI_PHY_STANDARD_80211b);
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                              "DataMode",StringValue (phyMode),
                              "ControlMode",StringValue (phyMode));
// Set it to adhoc mode
wifiMac.SetType ("ns3::AdhocWifiMac");
NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, c);

MobilityHelper mobility;
```

# 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



ns-3 NETWORK SIMULATOR

# 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)

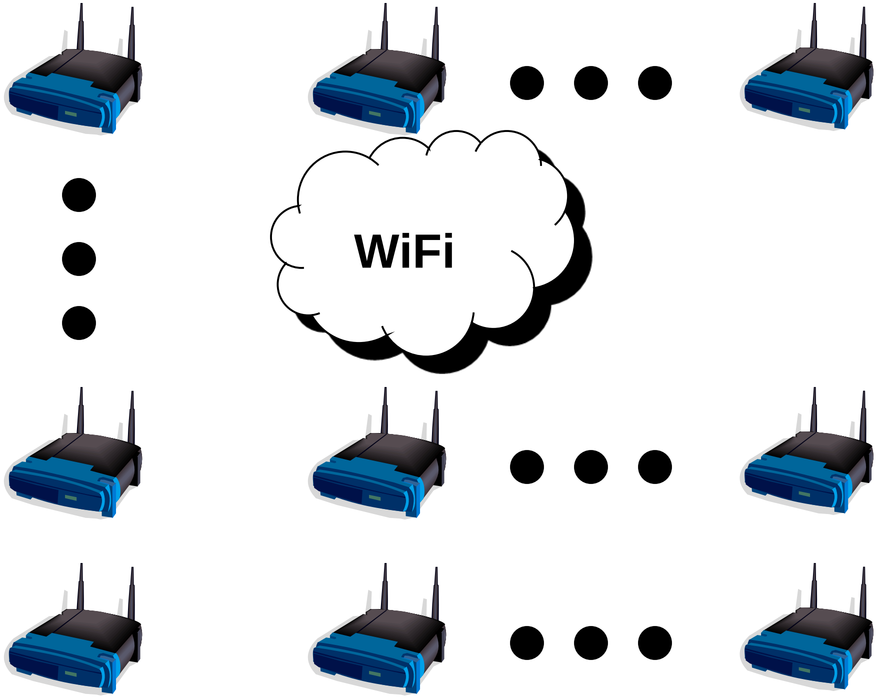
- Is not generic
- Does not try to allow code reuse
- Provides simple 'syntactical sugar' to make simulation scripts look nicer and easier to read for network researchers
- Each function applies a single operation on a ''set of same objects”
- A typical operation is "Install()"

# Helper Objects

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

# Example program

- (5x5) grid of WiFi ad hoc nodes
- OLSR packet routing
- Try to send packet from one node to another




Source (node 24) by default

- Let's look closely at how these objects are created

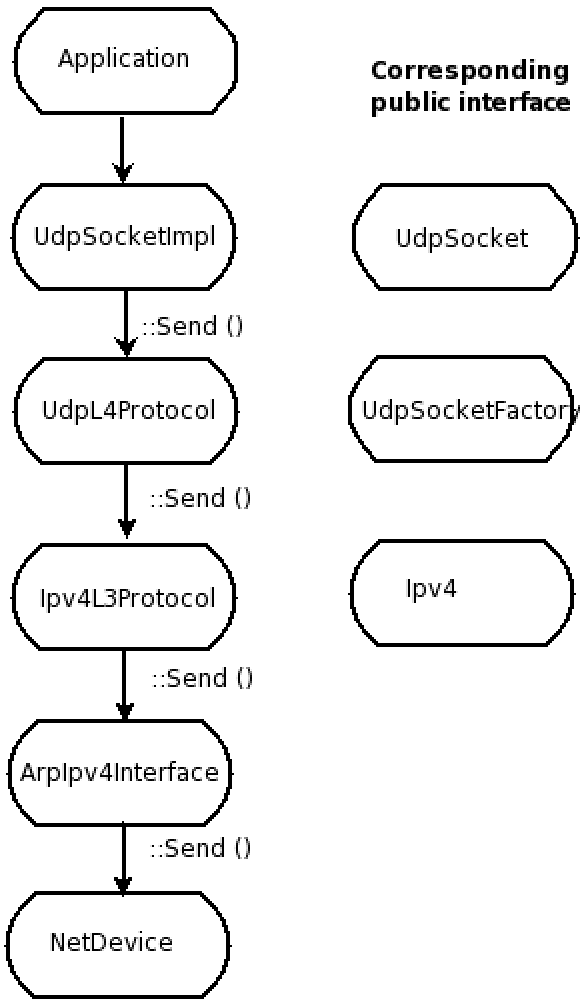
Sink (node 0) by default

# 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

NS-3 NETWORK SIMULATOR

# ns-3 TCP

- Several options exist:
  - native ns-3 TCP
    - Tahoe, Reno, NewReno (others in development)
  - TCP simulation cradle (NSC)
  - Use of virtual machines or DCE (more on this later)

- To enable NSC:

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

# ns-3 simulation cradle

- Port by Florian Westphal of Sam Jansen's Ph.D. work

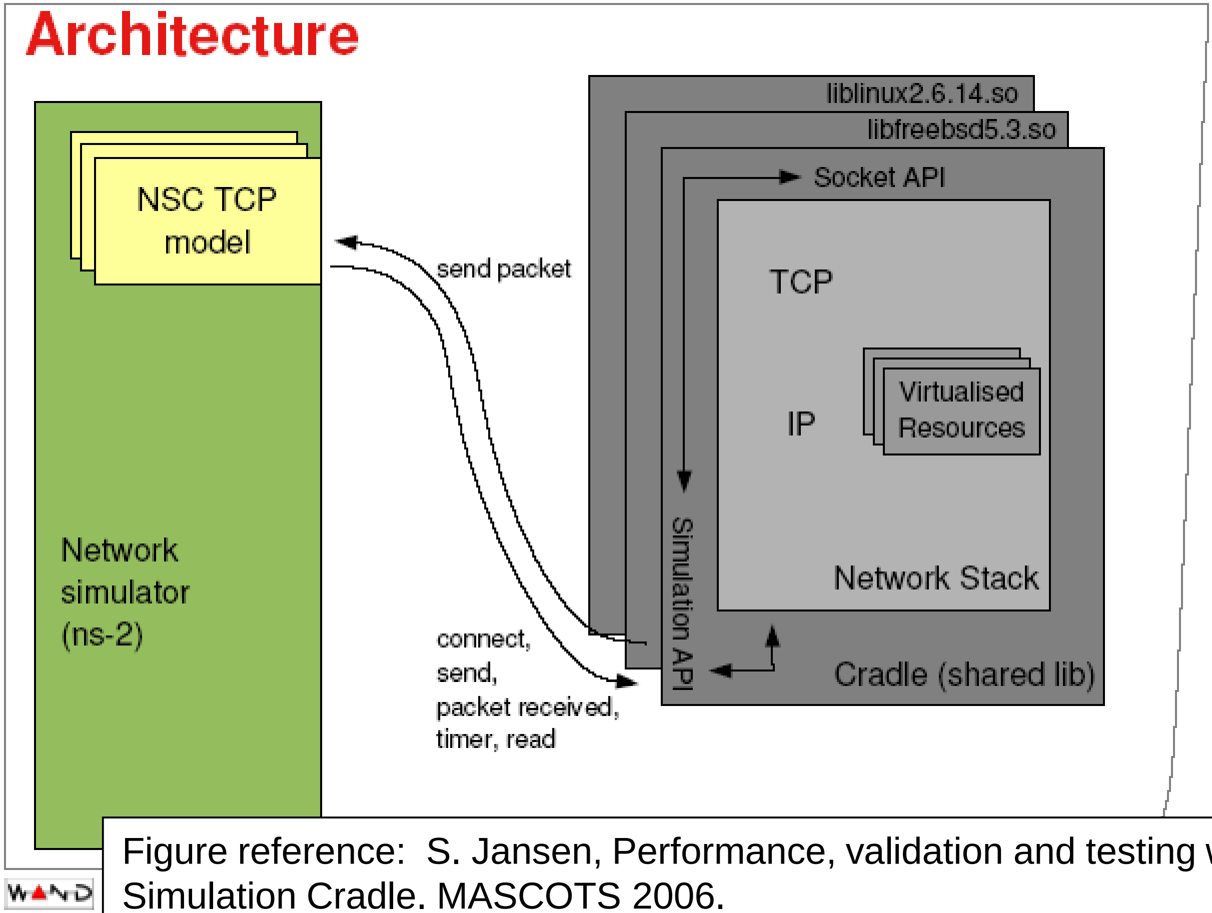
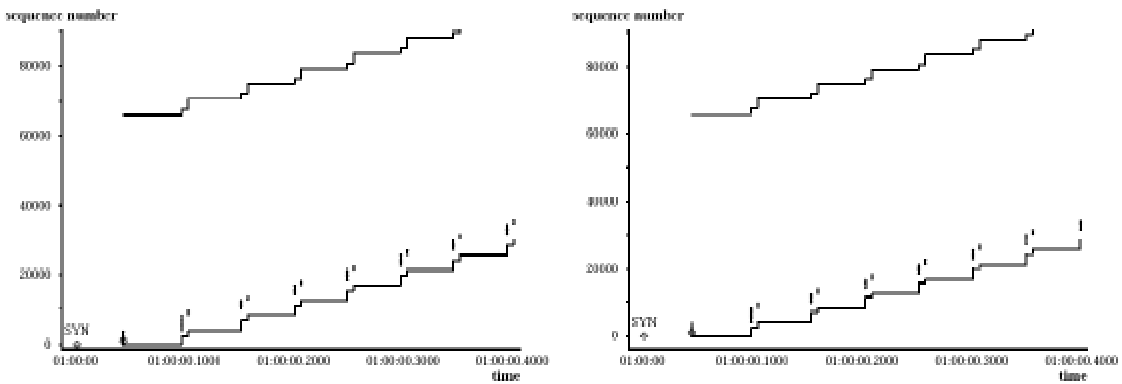



Figure reference: S. Jansen, Performance, validation and testing with the Network Simulation Cradle. MASCOTS 2006.

# ns-3 simulation cradle

## 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

For ns-3:
- Linux 2.6.18
- Linux 2.6.26
- Linux 2.6.28

Others:
- FreeBSD 5
- lwip 1.3
- OpenBSD 3

Other simulators:
- ns-2
- OMNET++

Figure reference: S. Jansen, Performance, validation and testing with the Network Simulation Cradle. MASCOTS 2006.

# IPv4 address configuration

- An Ipv4 address helper can assign addresses to devices in a NetDevice container

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

...

ipv4.NewNetwork ();  // bumps network to 10.1.2.0
otherCsmaInterfaces = ipv4.Assign (otherCsmaDevices);
```

# Applications and sockets

- In general, applications in ns-3 derive from the ns3::Application base class
  - A list of applications is stored in the ns3::Node
  - Applications are like processes
- Applications make use of a sockets-like API
  - Application::Start () may call ns3::Socket::SendMsg() at a lower layer

# Sockets API

## 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_ad
    dr);
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);
sendto(sk, ”hello”, 6, 0, (struct
    sockaddr *) &dest, sizeof(dest));
```

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

## ns-3 sockets

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

```
sk->Bind (InetSocketAddress (80));
```

```
sk->SendTo (InetSocketAddress (Ipv4Address
    (”10.0.0.1”), 80), Create<Packet>
    (”hello”, 6));
```

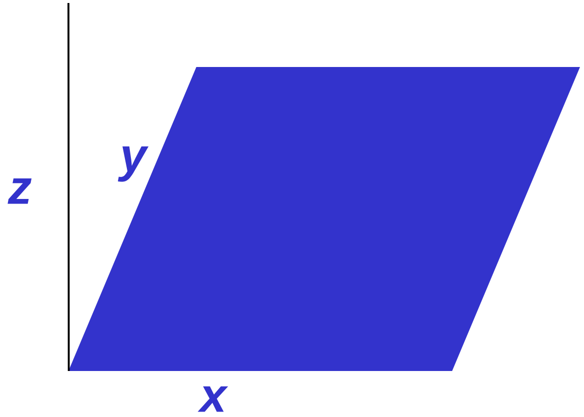
```
sk->SetReceiveCallback (MakeCallback
    (MySocketReceive));
•   […] (Simulator::Run ())

void MySocketReceive (Ptr<Socket> sk,
    Ptr<Packet> packet)
{
...
}
```



ns-3 NETWORK SIMULATOR

# Mobility models in ns-3

- The MobilityModel interface:
  - void SetPosition (Vector pos)
  - Vector GetPosition ()
- StaticMobilityModel
  - Node is at a fixed location; does not move on its own
- RandomWaypointMobilityModel
  - (works inside a rectangular bounded area)
  - 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
- RandomDirectionMobilityModel
  - 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




3D Cartesian coordinate system

# Object names

- It can be helpful to refer to objects by a string name
  - "access point"
  - "eth0"
- Objects can now be associated with a name, and the name used in the attribute system

# Names example

```
NodeContainer n;
n.Create (4);
Names::Add ("client", n.Get (0));
Names::Add ("server", n.Get (1));
...

Names::Add ("client/eth0", d.Get (0));
...

Config::Set ("/Names/client/eth0/Mtu", UintegerValue
   (1234));
```

Equivalent to:

```
Config::Set ("/NodeList/0/DeviceList/0/Mtu", UintegerValue
   (1234));
```

ns-3 NETWORK SIMULATOR

# 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

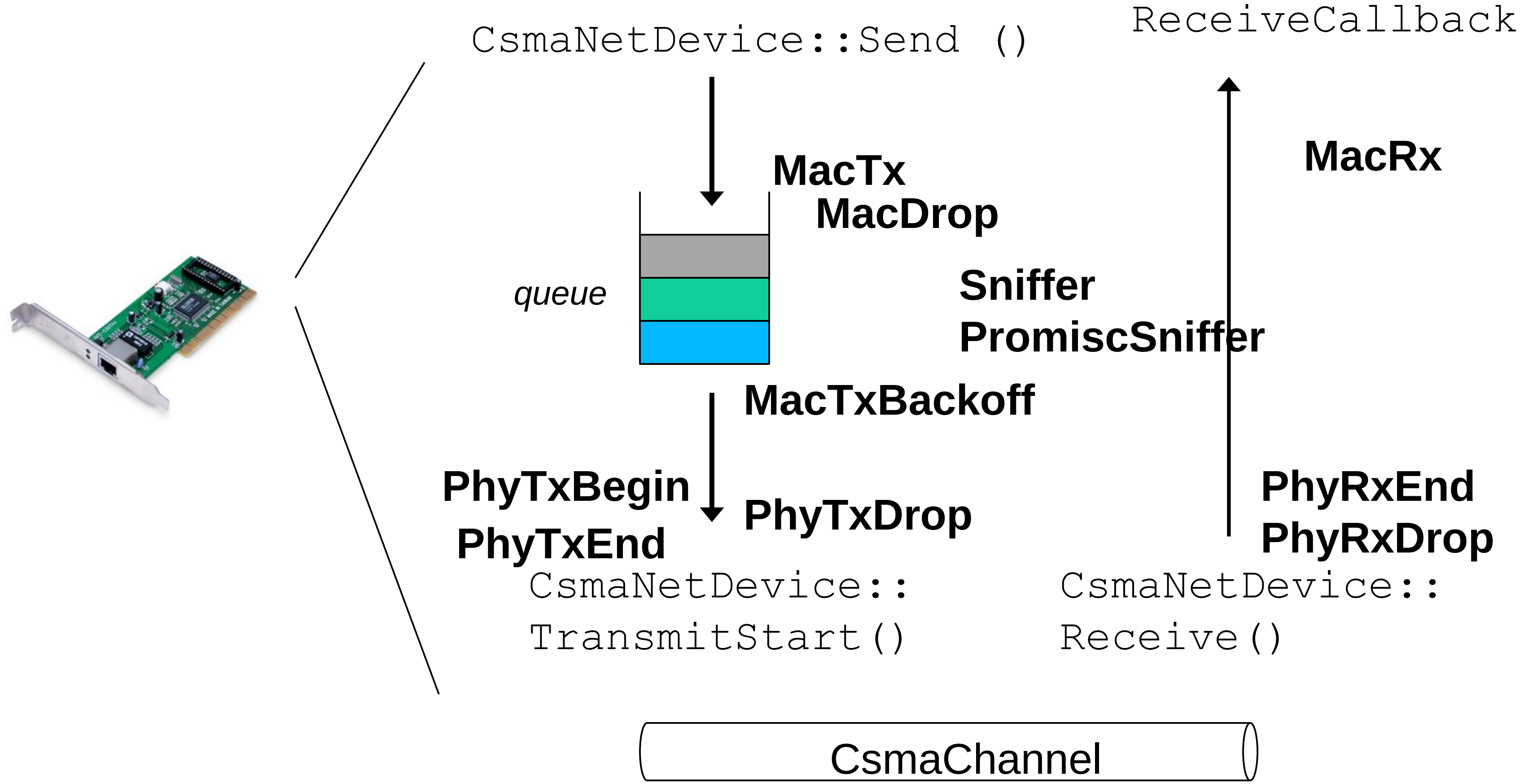
TracedValue → 
```
Config::Connect ("/path/to/traced/value", callback1);
```

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

TraceSource *unattached*

# NetDevice trace hooks

- Example: CsmaNetDevice




ns-3
NETWORK SIMULATOR