NS-3: Network Simulator 3

Gustavo J. A. M. Carneiro

INESC Porto / Faculdade de Engenharia / Universidade do Porto

UTM Lab Meeting – 2010-04-20
Outline

- NS-3 general overview
- NS-3 internal APIs overview
- Short Tutorial
Introduction: NS-2

- The most used simulator for network research
  - “Over 50% of ACM and IEEE network simulation papers from 2000-2004 cite the use of *ns-2*”
- Went unmaintained for a long period of time
- Outdated code design
  - Does not take into account modern programming
    - Smart pointers?
    - Design patterns?
  - Does not scale as well as some alternatives
    - (e.g. GTNetS)
  - Tracing system is difficult to use
    - Need to parse trace files to extract results
    - Trace files end up either
      - Having information researchers do not need, or
      - Missing information
        - It's usual practice to add printf's in the ns-2 code
Introduction: NS-3

- NS-3 is a **new** simulator, written *from scratch*
  - Not really an evolution of NS-2
- Programming languages: C++, Python
  - Unlike NS-2, everything designed for C++
  - Optional Python scripting
- Project started around mid 2006
  - Still under heavy development
- Official funded partners:
  - University of Washington
    - (Tom Henderson, Craig Dowell)
  - INRIA, Sophia Antipolis
    - (Mathieu Lacage)
  - Georgia Tech University (Atlanta)
    - George Riley (main author of *GTNetS*)
    - Raj Bhattacharjea
NS-3 Modules

- **Routing**: olsr, global-routing
- **Internet-stack**: (ipv4 impl.)
- **Devices**: csma | wifi | ...
- **Node class**: NetDevice ABC
- **Address types**: (IPv4, MAC, etc.)
- **Queues**: Socket ABC
- **IPv4 ABCs**: Packet Sockets
- **Helper**: High-level wrappers for everything else.
- **No smart pointers used.**
- **Aimed at scripting.**
- **Events**: Scheduler
- **Time arithmetic**:
- **Common**:
- **Node**:
- **Mobility**:
- **Simulator**:
- **Core**:
- **Smart pointers**: Dynamic type system
- **Attributes**: Callbacks, Tracing
- **Logging**: Random Variables
- **Packets**:
- **Packet Tags**:
- **Packet Headers**:
- **Pcap/ascii file writing**
- **Mobility Models**: (static, random walk, etc.)
Interesting NS-3 Features

- **Scalability features**
  - Packets can have "virtual zero bytes" (or dummy bytes)
    - For dummy application data that we don't care about
    - No memory is allocated for virtual zero bytes
    - Reduces the memory footprint of the simulation
  - Nodes have optional features (sort of AOP)
    - No memory waste in IPv4 stack for nodes that don't need it
    - Mobility model may not be needed
      - E.g. wired netdevices do not need to know the node position at all
    - New features can be easily added in the future
      - For example, energy models

- **Cross-layer features**
  - **Packet Tags**
    - Small units of information attached to packets
  - **Tracing**
    - Allow to report events across non-contiguous layers
Real-world Integration Features

- Real world integration features
  - Packets can be saved to **PCAP** files, in a real format
    - Many tools can read PCAP files, e.g. **Wireshark**
  - **Real-time scheduler**
    - Simulation events synchronized to ”wall clock time”
  - **”Network Simulation Cradle”**
    - Run Linux Kernel TCP/IP stack under simulation
      - Linux 2.6.18, Linux 2.6.26
  - **POSIX Emulation** (experimental)
    - Run unmodified POSIX programs under simulation
      - Special ELF loader converts POSIX API calls into NS-3 calls
    - Running routing daemons on NS-3 (planned)
Real world integration: EmuNetDevice
Visualization Status

- Visualization
  - Still experimental
  - Several ongoing attempts, none yet integrated
    - Example: ns-3-pyviz
      - (demoed in SIGCOMM workshop, Aug. 2008)
Distributed Simulation using MPI

- MPI: Message Passing Interface
  - Library (and protocol) for distributed applications
- New in NS-3.8, an MPI Simulator
  - Nodes in the simulation assigned different System Ids
  - Nodes with different System Ids run on different cluster machines
  - Nodes on different machines may communicate using point-to-point links only

Cluster Host 1

Node 0
SystemId 1

Node 1
SystemId 1

Node 2
SystemId 1

Cluster Host 2

Node 3
SystemId 2

Node 4
SystemId 2

Node 5
SystemId 2

Point-to-point Link (simulation and real world)
Link layer models

- Point-to-point (PPP links)
- Csma (Ethernet links)
- Bridge: 802.1D Learning Bridge
- Wifi (802.11 links)
  - EDCA QoS support (but not HCCA)
  - Both infrastructure (with beacons), and adhoc modes
- Mesh
  - 802.11s (but no legacy 802.11 stations supported yet)
  - ”Flame”: Forwarding LAyer for MEshing protocol
    - ”Easy Wireless: broadband ad-hoc networking for emergency services”
- Wimax: 802.16 (new in NS 3.8)
  - ”supports the four scheduling services defined by the 802.16-2004 standard”
- Tap-bridge, emu: testbed integration
Routing

- Adhoc:
  - OLSR (RFC 3626)
    - Since NS 3.8 with full HNA support (thanks Latih Suresh)
  - AODV (RFC 3561)
- "Global routing" (aka GOD routing)
  - Just computes static routes on simulation start
- Nix-vector Routing
  - Limited but high performance static routing
  - For simulations with thousands of wired nodes
- List-routing
  - Joins multiple routing protocols in the same node
  - For example: static routing tables + OLSR + AODV
Applications (traffic generators)

- **Onoff**
  - Generates streams, alternating on-and-off periods
  - Highly parameterized
    - Can be configured to generate many types of traffic
      - E.g. OnTime=1 and OffTime=0 means CBR
    - Works with either UDP or TCP
- **Packet sink**: receives packets or TCP connections
- **Ping6, v4ping**: send ICMP ECHO request
- **Udp-client/server**: sends UDP packet w/ sequence number
- **Udp-echo**: sends UDP packet, no sequence number
- **Radvd**: router advertisement (for IPv6)
NS 3: Performance


- "One sending node generates one packet every second and broadcasts it to its neighbors"
- "The neighboring nodes relay unseen messages after a delay of one second, thus flooding the entire network."
NS-3 Performance: Time

NS-3 Performance: Memory

(Preliminary) Conclusions

- NS-3 contains innovative and useful features
  - Scalable
  - Flexible
  - Clean design
  - Real-world (e.g. testbed) integration
- NS-3 has good performance
  - One of the fastest simulators around
  - The most memory efficient simulator around
- However
  - Not many models available for NS-3 yet
  - No GUI to build topology
  - Visualization still experimental
NS-3 internal APIs overview
Simulator Core

- Time is not manipulated directly: the Time class
  - Time class supports high precision 128 bit time values (nanosecond precision)

```cpp
Time t1 = Seconds (10);
Time t2 = t1 + MilliSeconds (100);
std::cout << t2.GetSeconds () << std::endl; // t2 = 10.1
```

- Get current time:
  - Time now = Simulator::Now ();

- Schedule an event to happen in 3 seconds:
  - void MyCallback (T1 param1, T2 param2) {...}
  - Simulator::Schedule (Seconds (3), MyCallback, param1, param2);
    - Values param1 and param2 passed as callback parameters
  - Also works with instance methods:
    Simulator::Schedule (Seconds (3), &MyClass::Method, instancePtr, param1, param2);
Random Variables

- 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, ...

```python
import numpy as np
import matplotlib.pyplot as plt

mu = 10.0
sigma = 5.0
rng = np.random.normal(mu, sigma, size=100000)
plt.hist(rng, bins=100)
plt.show()
```
Memory Management

- Many NS-3 objects use automatic garbage collection
- Reference counting
  - `Packet *p = new Packet;` # refcount initialized to 1
    p->Ref (); # refcount becomes 2
    p->Unref (); # refcount becomes 1
    p->Unref (); # refcount becomes 0, packet is freed
- Smart pointers
  - Manual reference counting is error prone
    - Can easily lead to memory errors
  - Smart pointers
    - Take care of all the reference counting work
    - Otherwise they behave like normal pointers
- Example:

```c++
void MyFunction () {
    Ptr<Packet> p = Create<Packet> (10);
    std::cerr << "Packet size: " << p->GetSize () << std::endl;
} # Packet is released (smart pointer goes out of scope)
```
Packets

- Packet objects used *vertically* in NS-3 to represent:
  - Units of information sent and received by applications
  - Information chunks of what will become a real packet
    (similar sk_buff in Linux kernel)
  - Simulated packets and L2/L1 frames being transmitted

- **Basic Usage**
  - **Create empty packet**
    - `Ptr<Packet> packet = Create<Packet> ();`
  - **Create packet with 10 ”dummy” bytes**
    - `Ptr<Packet> packet = Create<Packet> (10);`
    - ”Dummy” bytes are simulated as being there, but do not
      actually occupy any memory (reduces memory footprint)
  - **Create packet with user data**
    - `Ptr<Packet> packet = Create<Packet> ("hello", 5);`
  - **Copy a packet**
    - `Ptr<Packet> packet2 = packet1->Copy ();`
    - Note: packet copy is usually cheap (copy-on-write)
Nodes

- Node class
  - Represents a network element
  - **May** have an *IPv4 stack* object
    - But it is completely optional!
  - **May** have a *mobility model*
    - But it is optional, e.g. CsmaNetDevice needs no mobility model
  - Contains a list of *NetDevices*
  - Contains a list of *Applications*

- NodeList class (singleton)
  - Tracks all nodes ever created
  - Node index <=> Ptr conversions
uint64_t g_packetDrops = 0;
uint64_t g_packetDropBytes = 0;

void TraceDevQueueDrop (std::string context,
    Ptr<const Packet> droppedPacket)
{
    g_packetDrops += 1;
    g_packetDropBytes += droppedPacket->GetSize();
}

int main (int argc, char *argv[])
{
    [...] Config::Connect ("/NodeList/*/DeviceList/*/TxQueue/Drop",
            MakeCallback (&TraceDevQueueDrop));
    [...]
Packet: Headers and Trailers

- Packets support *headers* and *trailers*
  - Headers an trailers are implemented as classes that
    - Implement a **Serialize** method:
      - Writes the header information as a byte stream;
    - Implement a **Deserialize** method:
      - Reads the header information from a byte stream;
- Headers and trailers used to implement protocols
- Packets contain exact byte contents
  - They are not just structures as in NS-2
  - Allows writing pcap trace files, readable from wireshark
- **LLC/SNAP example (from ns-3):**

  ```cpp
  uint32_t LlcSnapHeader::GetSerializedSize (void) const
  {
    return 1 + 1 + 1 + 3 + 2;
  }

  void LlcSnapHeader::Serialize (Buffer::Iterator start) const
  {
    Buffer::Iterator i = start;
    uint8_t buf[] = {0xa0, 0xa0, 0x03, 0, 0, 0};
    i.Write (buf, 6);
    i.WriteHtonU16 (m_etherType);
  }

  uint32_t LlcSnapHeader::Deserialize (Buffer::Iterator start) const
  {
    Buffer::Iterator i = start;
    i.Next (5+1);   // skip 6 bytes, don't care about content
    m_etherType = i.ReadNtohU16 ();
    return GetSerializedSize ();
  }
  ```

- **Adding a header:**
  - LlcSnapHeader llcsnap;
    llcsnap.SetType (0x0800);  # Ipv4
    packet->AddHeader (llcsnap);

- **Removing a header:**
  - LlcSnapHeader llcsnap;
    if (packet->RemoveHeader (llcsnap) {  
      std::cout << llcsnap.GetType () << std::endl;
    }
Callback Objects

- NS-3 Callback class implements function objects
  - Type safe callbacks, manipulated by value
    - Used for example in sockets and tracing

- Example

```java
double MyFunc (int x, float y) {
    return double (x + y) / 2;
}

[...]
Callback<double, int, float> cb1;
cb1 = MakeCallback (MyFunc);
double result = cb1 (2, 3); // result receives 2.5

[...]
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); // result receives 2.5
```
NS-3 Sockets

- Plain C sockets

```c
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);
sendto(sk, "hello", 6, 0, (struct sockaddr *) &dest, sizeof(dest));

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

- NS-3 sockets

```c
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)
{
...
}
```
Tutorial
int main (int argc, char *argv[]) {
    LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);

    NodeContainer nodes;
    nodes.Create (2);
}
PointToPointHelper pointToPoint;

pointToPoint.SetDeviceAttribute("DataRate", StringValue("5Mbps"));

pointToPoint.SetChannelAttribute("Delay", StringValue("2ms"));

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

Ipv4AddressHelper `address`;
`address.SetBase ("10.1.1.0", "255.255.255.0")`;

Ipv4InterfaceContainer `interfaces = address.Assign (devices)`;

```
interface: Ipv4InterfaceContainer
```

```
10.1.1.1
```

```
Node 0
```

```
ipv4Interface

Node 1
```

```
10.1.1.2
```

```
PointToPointNetDevice

```
UdpEchoServerHelper echoServer (9);  
ApplicationContainer serverApps =  
  echoServer.Install (nodes.Get (1));  
serverApps.Start (Seconds (1.0));  
serverApps.Stop (Seconds (10.0));
UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps =
echoClient.Install (nodes.Get (0));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));

clientApps: ApplicationContainer

Dest: 10.1.1.2, port 9
1 packet, 1024 bytes

UdpEchoClient

Node 0

UdpEchoServer

Node 1
```cpp
[...
    Simulator::Run ();
    Simulator::Destroy ();
    return 0;
}
```

```bash
$ ./waf --run first
[...]
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```
import ns3

ns3.LogComponentEnable("UdpEchoClientApplication", ns3.LOG_LEVEL_INFO)
ns3.LogComponentEnable("UdpEchoServerApplication", ns3.LOG_LEVEL_INFO)

nodes = ns3.NodeContainer()
nodes.Create(2)

pointToPoint = ns3.PointToPointHelper()
pointToPoint.SetDeviceAttribute("DataRate", ns3.StringValue("5Mbps"))
pointToPoint.SetChannelAttribute("Delay", ns3.StringValue("2ms"))

devices = pointToPoint.Install(nodes)

stack = ns3.InternetStackHelper()
stack.Install(nodes)

address = ns3.Ipv4AddressHelper()
address.SetBase(ns3.Ipv4Address("10.1.1.0"), ns3.Ipv4Mask("255.255.255.0"))
interfaces = address.Assign(devices)

echoServer = ns3.UdpEchoServerHelper(9)

serverApps = echoServer.Install(nodes.Get(1))
serverApps.Start(ns3.Seconds(1.0))
serverApps.Stop(ns3.Seconds(10.0))

echoClient = ns3.UdpEchoClientHelper(interfaces.GetAddress(1), 9)
echoClient.SetAttribute("MaxPackets", ns3.UintegerValue(1))
echoClient.SetAttribute("Interval", ns3.TimeValue(ns3.Seconds(1.0)))
echoClient.SetAttribute("PacketSize", ns3.UintegerValue(1024))

clientApps = echoClient.Install(nodes.Get(0))
clientApps.Start(ns3.Seconds(2.0))
clientApps.Stop(ns3.Seconds(10.0))

ns3.Simulator.Run()
nns3.Simulator.Destroy()
import sys
import ns3

DISTANCE = 150  # (m)
NUM_NODES_SIDE = 3

def main(argv):
    cmd = ns3.CommandLine()

    cmd.NumNodesSide = None
    cmd.AddValue("NumNodesSide", "Grid side number of nodes (total number of nodes will be this number squared)")

    cmd.Results = None
    cmd.AddValue("Results", "Write XML results to file")

    cmd.Plot = None
    cmd.AddValue("Plot", "Plot the results using the matplotlib python module")

    cmd.Parse(argv)
wifi-olsr-flowmon.py (2/8)

(...continued from main...)

wifi = ns3.WifiHelper.Default()
wifiMac = ns3.NqosWifiMacHelper.Default()
wifiPhy = ns3.YansWifiPhyHelper.Default()
wifiChannel = ns3.YansWifiChannelHelper.Default()
wifiPhy.SetChannel(wifiChannel.Create())
wifi.SetRemoteStationManager("ns3::ArfWifiManager")
wifiMac.SetType("ns3::AdhocWifiMac",
internet = ns3.InternetStackHelper()
list_routing = ns3.Ipv4ListRoutingHelper()
olsr_routing = ns3.OlsrHelper()
static_routing = ns3.Ipv4StaticRoutingHelper()
list_routing.Add(static_routing, 0)
list_routing.Add(olsr_routing, 100)  # OLSR takes precedence!
internet.SetRoutingHelper(list_routing)

ipv4Addresses = ns3.Ipv4AddressHelper()
ipv4Addresses.SetBase(ns3.Ipv4Address("10.0.0.0"),
                       ns3.Ipv4Mask("255.255.255.0"))
port = 9  # Discard port(RFC 863)
onOffHelper = ns3.OnOffHelper("ns3::UdpSocketFactory",
ns3.Address(ns3.InetSocketAddress(ns3.Ipv4Address("10.0.0.1"), port)))

onOffHelper.SetAttribute("DataRate",
ns3.DataRateValue(ns3.DataRate("100kbps")))

onOffHelper.SetAttribute("OnTime",
ns3.RandomVariableValue(ns3.ConstantVariable(1)))

onOffHelper.SetAttribute("OffTime",
ns3.RandomVariableValue(ns3.ConstantVariable(0)))
addresses = []
nodes = []

# C++: for (int xi = 0; xi < num_nodes_side; xi++) {
for xi in range(num_nodes_side):
    # C++: for (int yi = 0; yi < num_nodes_side; yi++) {
        for yi in range(num_nodes_side):

            node = ns3.Node()
nodes.append(node)

            mobility = ns3.ConstantPositionMobilityModel()
mobility.setPosition(ns3.Vector(xi*DISTANCE, yi*DISTANCE, 0))
node.AggregateObject(mobility)

devices = wifi.Install(wifiPhy, wifiMac, node)

internet.Install(node)  # adds Ipv4 and static+OLSR routing
ipv4_interfaces = ipv4Addresses.Assign(devices)
addresses.append(ipv4_interfaces.GetAddress(0))
for i, node in enumerate(nodes):
    destaddr = addresses[(len(addresses) - 1 - i) % len(addresses)]
onOffHelper.SetAttribute("Remote",
    ns3.AddressValue(ns3.InetSocketAddress(destaddr, port)))
app = onOffHelper.Install(ns3.NodeContainer(node))

flowmon_helper = ns3.FlowMonitorHelper()
monitor = flowmon_helper.InstallAll()
monitor.SetAttribute("DelayBinWidth", ns3.DoubleValue(0.001))
monitor.SetAttribute("JitterBinWidth", ns3.DoubleValue(0.001))
monitor.SetAttribute("PacketSizeBinWidth", ns3.DoubleValue(20))

ns3.Simulator.Stop(ns3.Seconds(44.0))
ns3.Simulator.Run()

classifier = flowmon_helper.GetClassifier()

if cmdPlot is not None: # if --Plot cmdline option given:
    import pylab
    delays = []
    for flow_id, flow_stats in monitor.GetFlowStats():
        # filter out UDP port 698 (OLSR)
        tupl = classifier.FindFlow(flow_id)
        if tupl.protocol == 17 and tupl.sourcePort == 698:
            continue

        delays.append(flow_stats.delaySum.GetSeconds() / flow_stats.rxPackets)

pylab.hist(delays, 20)
pylab.xlabel("Delay (s)")
pylab.ylabel("Number of Flows")
pylab.show()
wifi-olsr-flowmon.py (8/8)
Questions?
Mobility Models

- 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
Getting Started: Linux

- Building it
  1) `sudo apt-get install build-essential g++ python mercurial` # (Ubuntu)
  2) `hg clone http://code.nsnam.org/ns-3-allinone/`
  3) `cd ns-3-allinone`
  4) `./download.py` # will download components
  5) `./build.py` # will build NS-3
  6) `cd ns-3-dev`

- Running example programs
  - Programs are built as
    `build/<variant>/path/program_name`
    - `<variant>` is either debug or optimized
  - Using `waf --shell`
    1) `./waf --shell`
    2) `./build/debug/examples/simple-point-to-point`
  - Using `waf --run`
    1) `./waf --run simple-point-to-point`
Getting Started: Windows

- Building it
  1) Install build tools
     1) Cygwin or Mingw GCC (g++)
     2) Python: http://www.python.org
     3) Mercurial: http://mercurial.berkwood.com/
  2) hg clone http://code.nsnam.org/ns-3.0.11/
  3) cd ns-3.0.11
  4) waf configure # optional: -d optimized
  5) waf check # runs unit tests

- Rest of instructions the same as in Linux...
Packet: Tags

- Tags
  - Small chunks of information
  - Any number of tags can be attached a packet
  - Tags are keyed by the a structure type itself
    - `Ptr<Packet> p;`
    - `MyTag tag;`
    - `p->AddTag (tag)`
    - `p->PeekTag (tag)`
  - New tag types are defined similarly to header types

- Tags can be used to:
  - Attach context information to a packet
    - Example: NetDevice attaches destination MAC address when queueing, retrieves it when dequeuing for transmission
  - Convey additional information across layers
class Object

- **Object** is the base class for many important classes:
  - Node, NetDevice, Application, Socket, ...
- **class Object** provides many useful features
  - Basic memory management (reference counting)
  - Advanced memory management (the Dispose method)
    - Dispose/DoDispose: used to break reference counting loops
      - Node => list(Application); Application => Node
- **Object aggregation**
  - COM-like interface query mechanism
  - Instead of a huge class, split class into several objects:
    - Node, Ipv4, [Udp/Tcp]SocketFactory, Mobility,...
  - Example: from a Node object, see if it supports Ipv4
- void MyFunction (Ptr<Node> node)
  {
    Ptr<Ipv4> ipv4 = node->GetObject<Ipv4> ();
    if (ipv4 != NULL)
      std::cerr << "Node has " << ipv4->GetNRoutes ()
                << " routes." << std::endl;
  }
- **Tracing hooks**
Object and TypeId

- **TypeId**: working around C++ limitations
  - In C++, classes are not *first-class objects*
- **TypeId** is an object that describes a class type:
  - Type name
  - List of *attributes* or *trace sources*
- **TypeId** implements the *Factory* Design Pattern
  - Example: to create an object from type name:
    ```cpp
    TipoId objType = TipoId::LookupByName ("StaticMobilityModel")
    Ptr<Object> mobilityModel = objType.CreateObject ()
    ```
Object and TypeId (cont.)

- Because of the TypeId system, creating Object instances should be done with:
  - `Ptr<ClassName> obj = CreateObject<ClassName>(...parameters)`

- Defining new Object subclasses needs special care:
  - Must define a GetTypeIds static method, like this:

```cpp
class MyClass : public MyParent
{
  public:
    MyClass (ParamType1 p1, ...);
    static TypeId GetTypeIds (void);
  
} [
};

TypeId
MyClass::GetTypeIds (void)
{
  static TypeId tid = TypeId ("MyClass")
  .SetParent<MyParent> ()
  .AddConstructor<MyClass, ParamType1, ...> ()
  return tid;
}"
```
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
Applications and Sockets

- Each Node contains a list of Applications
  - Applications are like *processes* in a normal system
- Applications contain a number of Sockets
  - Sockets represent communication end points
  - NS-3 sockets modelled after the BSD socket API
- Example uses of Applications
  - Traffic generators (e.g. OnOffApplication)
  - Traffic sinks (e.g. to respond to connection requests)
  - Routing agents, higher level protocols
    - Whatever normally runs in userspace in a UNIX system
- Sockets creation: a *socket factory* Node interface:

```cpp
Ptr<SocketFactory> udpFactory = node->GetObject<SocketFactory>(
  (TypeId::LookupByName("Udp")));
Ptr<Socket> socket = udpFactory->CreateSocket();
```