ns-3 Training

ns-3 training, June 2017
Simulator core

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

Execute a function (may generate additional events)

Advance the virtual time to the next event (function)
Simulator example

```cpp
#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
# 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

1. Handle program inputs
2. Configure topology
3. Run simulation
4. Process outputs
Command-line arguments

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

```c
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
  – Note: Changing resolution is not well used/tested

• Time objects can be set by floating-point values and can export floating-point values
  
  \[
  \text{double timeDouble} = \text{t.GetSeconds}();
  \]
  – Best practice is to avoid floating point conversions where possible
Events in ns-3

• Events are just function calls that execute at a simulated time
  – i.e. callbacks
  – this is 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

- 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
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
  – Period of PRNG is $3.1 \times 10^{57}$
• Partitions a pseudo-random number generator into uncorrelated *streams* and *substreams*
  – Each RandomVariableStream gets its own stream
  – This stream partitioned into substreams
Key Terminology

• **Seed**: A set of values that generates an entirely new PRNG sequence

• **Stream**: The PRNG sequence is divided into non-overlapping intervals called streams

• **Run Number (substream)**: Each stream is further divided to substreams, indexed by a variable called the run number.
Streams and Substreams

Incrementing the Run Number will move all streams to a new substream.

Each ns-3 RandomVariableStream object is assigned to a stream (by default, randomly).

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 uncorrelated streams
Setting the stream number

- The ns-3 implementation provides access to $2^{64}$ streams
- $2^{63}$ are placed in a pool for automatic assignment, and $2^{63}$ are reserved for fixed assignment

• Users may optionally assign a stream number index to a random variable using the `SetStream()` method.
  - This allows better control over selected random variables
  - Many helpers have `AssignStreams()` methods to do this across many such random variables
Putting it together

• Example of scheduled event

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

```cpp
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 Training

Program Structure and Simulation Campaigns
Example walkthrough

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

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

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

Nodes contain Applications, “stacks”, and NetDevices
Node basics

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

"DTN"
(Originally) NetDevices were strongly bound to Channels of a matching type

- ns-3 Spectrum models relax this assumption

Nodes are architected for multiple interfaces
Internet Stack

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

- Devices
  - WiFi, WiMAX, CSMA, Point-to-point, ...
- 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

```c
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
    // Handle any post-simulation data processing
}
```
Helper API

• The ns-3 “helper API” provides a set of classes and methods that make common operations easier than using the low-level API

• Consists of:
  – container objects
  – helper classes

• The helper API is implemented using the low-level API

• Users are encouraged to contribute or propose improvements to the ns-3 helper API
Containers

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

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

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

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

NodeContainer c;
c.Create (numNodes);
...
... mobility.Install (c);
...
... internet.Install (c);
...
Native IP models

- IPv4 stack with ARP, ICMP, UDP, and TCP
- IPv6 with ND, ICMPv6, IPv6 extension headers, TCP, UDP
- IPv4 routing: RIPv2, static, global, NixVector, OLSR, AODV, DSR, DSDV
- IPv6 routing: RIPvng, static
IP address configuration

• An Ipv4 (or Ipv6) address helper can assign addresses to devices in a NetDevice container

```cpp
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);
```
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.
ApplicationContainer apps;
OnOffHelper onoff ("ns3::UdpSocketFactory",
        InetSocketAddress ("10.1.2.2", 1025));
onoff.SetAttribute ("OnTime", StringValue ("Constant:1.0"));
onoff.SetAttribute ("OffTime", StringValue ("Constant:0.0"));
apps = onoff.Install (csmaNodes.Get (0));
apps.Start (Seconds (1.0));
apps.Stop (Seconds (4.0));

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));
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
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));

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

ns-3 training, June 2017
NetDevice trace hooks

- Example: CsmaNetDevice

  **CsmaNetDevice::Send()**

  **NetDevice::ReceiveCallback**

  **queue**

  MacTx
  MacDrop

  Sniffer
  PromiscSniffer

  MacTxBackoff

  PhyTxBegin
  PhyTxEnd

  PhyTxDrop

  **CsmaNetDevice::TransmitStart()**

  **CsmaNetDevice::Receive()**

  **CsmaChannel**
LTE/Wi-Fi Coexistence

case study
Use case: LAA Wi-Fi Coexistence

- ns-3 has been extended to support scenarios for LTE LAA/Wi-Fi Coexistence
- Methodology defined in 3GPP Technical Report TR36.889
- Enhancements needed:
  - Wireless models (LBT access manager, SpectrumWifiPhy, propagation/fading models)
  - Scenario support (traffic models)
  - Output data processing
Indoor 3GPP scenario

Step 1: Both operators A and B are Wi-Fi co-channel on separate SSID
Step 2: (depicted) Replace operator A network with LTE LAA

5 UEs/STAs per cell per operator (40 total) randomly dropped
4 co-channel cells per operator (eNB or Wi-Fi AP)
Downlink on shared channel; LAA has separate licensed uplink

Idealized backhaul to traffic sources

ns-3 training, June 2017
## Indoor scenario details

<table>
<thead>
<tr>
<th>Parameter</th>
<th>3GPP TR 36.889</th>
<th>ns-3 implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Network Layout</strong></td>
<td>Indoor scenario</td>
<td>Indoor scenario</td>
</tr>
<tr>
<td><strong>System bandwidth</strong></td>
<td>20 MHz</td>
<td>20 MHz</td>
</tr>
<tr>
<td><strong>Carrier frequency</strong></td>
<td>5 GHz</td>
<td>5 GHz (channel 36, tunable)</td>
</tr>
<tr>
<td><strong>Number of carriers</strong></td>
<td>1, 4 (to be shared between two operators)</td>
<td>1 for evaluations with DL+UL Wi-Fi coexisting with DL-only LAA</td>
</tr>
<tr>
<td><strong>Total Base Station (BS) transmission power</strong></td>
<td>18/24 dBm</td>
<td>18/24 dBm</td>
</tr>
<tr>
<td><strong>Total User equipment (UE) transmission power</strong></td>
<td>18 dBm</td>
<td>Simulations herein consider 18 dBm</td>
</tr>
<tr>
<td><strong>Distance dependent path loss, shadowing and fading</strong></td>
<td>ITU InH</td>
<td>802.11ax indoor model</td>
</tr>
<tr>
<td><strong>Antenna pattern</strong></td>
<td>2D Omni-directional</td>
<td>2D Omni-directional</td>
</tr>
<tr>
<td><strong>Antenna height</strong></td>
<td>8 m</td>
<td>8 m (LAA, not modelled for Wi-Fi)</td>
</tr>
<tr>
<td><strong>UE antenna height</strong></td>
<td>1.6 m</td>
<td>1.5 m (LAA, not modelled for Wi-Fi)</td>
</tr>
<tr>
<td><strong>Antenna gain</strong></td>
<td>5 dBi</td>
<td>5 dBi</td>
</tr>
<tr>
<td><strong>UE antenna gain</strong></td>
<td>0 dBi</td>
<td>0 dBi</td>
</tr>
<tr>
<td><strong>Number of UEs</strong></td>
<td>10 UEs per unlicensed band carrier per operator for DL-only 10 UEs per unlicensed band carrier per operator for DL-only for four unlicensed carriers 20 UEs per unlicensed band carrier per operator for DL+UL for single unlicensed carrier 20 UEs per unlicensed band carrier per operator for DL+UL Wi-Fi coexisting with DL-only LAA</td>
<td>Supports all the configurations in TR 36.889. Simulations herein consider the case of 20 UEs per unlicensed band carrier per operator for DL LAA coexistence evaluations for single unlicensed carrier.</td>
</tr>
<tr>
<td><strong>UE Dropping</strong></td>
<td>All UEs should be randomly dropped and be within coverage of the small cell in the unlicensed band</td>
<td>Randomly dropped and within small cell coverage.</td>
</tr>
<tr>
<td><strong>Traffic Model</strong></td>
<td>FTP Model 1 and 3 based on TR 36.814 FTP model file size 0.5 Mbytes, Optional: VoIP model based on TR36.889</td>
<td>FTP Model 1 as in TR36.814, FTP model file size 0.5 Mbytes, Voice model: DL only</td>
</tr>
<tr>
<td><strong>UE noise figure</strong></td>
<td>9 dBA</td>
<td>9 dBA</td>
</tr>
<tr>
<td><strong>Cell selection</strong></td>
<td>For LAA UEs, cell selection is based on RSRP (Reference Signal Received Power) For WiFi stations (STAs), cell selection is based on RSS (Received signal power strength) of WiFi Access Points (APs). RSS threshold is -82 dBm.</td>
<td>RSRP for LAA UEs and RSS for Wi-Fi STAs</td>
</tr>
<tr>
<td><strong>Network synchronization</strong></td>
<td>For the same operator, the network can be synchronized. Small cells of different operators are not synchronized.</td>
<td>Small cells are synchronized, different operators are not synchronized.</td>
</tr>
</tbody>
</table>
Outdoor 3GPP scenario

Outdoor layout: hexagonal macrocell layout. 7 macro sites and 3 cells per site. 1 Cluster per cell. 4 small cells per operator per cluster, uniformly dropped. ITU UMi channel model.

Figure source: 3GPP TR 36.889 V13.0.0 (2015-05)
References

• ns-3 Wiki page:
  – https://www.nsnam.org/wiki/LAA-WiFi-Coexistence
    • module documentation
    • references to various publications
    • documentation on reproducing results

• Code:
  – http://code.nsnam.org/laa/ns-3-lbt
Sample results

These flows (the majority in the scenario) experience no contention and achieve close to the MCS 15 limit.

Some flows experience contention and slow down.

a) Step 1 (Wi-Fi)

b) Step 2 (LAA)

Small amount of throughput degradation for the non-replaced Wi-Fi network.
Gnuplot

- `src/tools/gnuplot.{cc,h}`
- C++ wrapper around gnuplot
- classes:
  - Gnuplot
  - GnuplotDataset
    - Gnuplot2dDataset, Gnuplot2dFunction
    - Gnuplot3dDataset, Gnuplot3dFunction
Enabling gnuplot for your code

- examples/wireless/wifi-clear-channel-cmu.cc

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

Gnuplot gnuplot = Gnuplot ("clear-channel.eps");
for (uint32_t i = 0; i < modes.size (); i++)
{
    std::cout << modes[i] << std::endl;
    Gnuplot2dDataset dataset (modes[i]);
}
```

produce a plot file that will generate an EPS figure

one dataset per mode

```cpp
uint32_t pktsRecvd = experiment.Run (wifi, wifiPhy, wifiMac, wifiChannel);
dataset.Add (rss, pktsRecvd);
}
gnuplot.AddDataset (dataset);
```

Add data to dataset

Add dataset to plot
Matplotlib

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

```python
# Demonstrate use of ns-3 as a random number generator integrated
# 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()
```
ns-3 Training: Packets

ns-3 Annual meeting June 2017
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
ns-3 Annual meeting June 2017
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 are serialized into the packet byte buffer with Packet::AddHeader() and removed with Packet::RemoveHeader()

• Headers can also be 'Peeked' without removal

```cpp
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
- Tags presently are not preserved across serialization boundaries (e.g. MPI)
PacketTag vs. ByteTag

- Two tag types are available: PacketTag and ByteTag
  - ByteTags run with bytes
  - PacketTags run with packets
- When Packet is fragmented, both copies of Packet get copies of PacketTags
- When two Packets are merged, only the PacketTags of the first are preserved
- PacketTags may be removed individually; ByteTags may be removed all at once
Here is a simple example illustrating the use of tags from the code in src/internet/model/udp-socket-impl.cc:

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

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

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

- Packets may optionally carry metadata
  - record every operation on a packet's buffer
  - implementation of Packet::Print for pretty-printing of the packet
  - sanity check that when a Header is removed, the Header was actually present to begin with

- Not enabled by default, for performance reasons

- To enable, insert one or both statements:
  ```cpp
  Packet::EnablePrinting ();
  Packet::EnableChecking ();
  ```
Ptr<Packet>

• Packets are reference counted objects that support the smart pointer class \( \text{Ptr} \)

• Use a templated "Create" method instead of CreateObject for ns3::Objects

• Typical creation:
  - \( \text{Ptr<Packet>} \text{pkt} = \text{Create<Packet> ()}; \)

• In model code, Packet pointers may be const or non-const; often Packet::Copy() is used to obtain non-const from const
  - \( \text{Ptr<\text{const Packet}> \text{cpkt} = \ldots}; \)
  - \( \text{Ptr<Packet>} \text{p} = \text{cpkt-} \text{Copy ()}; \)