Networks Simulation

Corso di Tecnologie di Infrastrutture di Reti

Carlo Augusto Grazia, Martin Klapez, Natale Patriciello

Department of Engineering *Enzo Ferrari*
University of Modena and Reggio Emilia

UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Modena, 27 May 2015

Overview

- Hands on ns-3
  - Precondition: Last lesson on ns-3 simulator
  - Today: Write C++ code to perform simulations
  - Postcondition: To be able to hack ns-3 (beginner level)
- *nix machine required, have you installed the software?
- We are three, you up to 20, let's do a great job togheter
- Do you want a mini-thesis or a thesis on Reti? It's better to pay attention :-)

Install ns3: Using git as source code manager

On Linux (he -- point to Nat -- hates Ubuntu):

```
$ git clone https://github.com/nsnam/ns-3-dev-git.git
$ cd ns-3-dev-git
$ ./waf configure --enable-examples
$ ./waf --run first
```

On Mac OS X

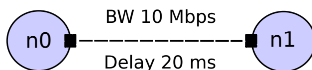
```
$ git clone https://github.com/nsnam/ns-3-dev-git.git
$ cd ns-3-dev-git
$ ./waf configure --enable-examples
$ ./waf --run first
```

Last Lesson..

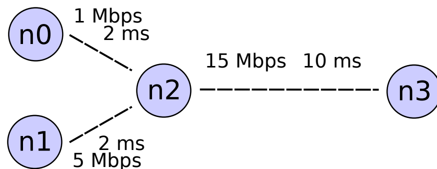
What has been shown:

Examples of simulations: included in sources, they had been ran and the pcap output has been shown

- TCP Bulk: Two nodes exchange data over a point-to-point



- Global routing: Two nodes compete for a bottleneck link using UDP



This Lesson..

What YOU will do today:

Learn how to “interact” with simulations, modify them to suit your needs, and create your own simulations

- Time
- Attributes
- Callbacks
- Traced Values

Time

Teacher..

You promised us exercises!

Let me introduce time.. *simulated* time. And before that, the core object of ns-3:

Example

What is the Simulator class? It is the public entry point to access event scheduling facilities. Once a couple of events have been scheduled to start the simulation, the user can start to execute them by entering the simulator main loop (call Simulator::Run). Once the main loop starts running, it will sequentially execute all scheduled events in order from oldest to most recent until there are either no more events left in the event queue or Simulator::Stop has been called.

Scheduling events

You can schedule events through Callbacks. The API is (MEM is a typedef which indicates a "function pointer", and OBJ indicates a pointer to an instance):

```
Simulator::Schedule (Time const &time, MEM mem_ptr, OBJ obj);
```

or

```
Simulator::ScheduleWithContext (uint32_t context,
        Time const &time, MEM mem_ptr, OBJ obj);
```

What is context?

Context is an advanced feature. In simple words, it is the node id on which the callback should be executed in.

Searching through API

Gathering more information on classes and API

I introduced the class Time. I can do a slide with its meaning, how to use it, and more...

BUT...

I'm here to teach you on HOW to gather information yourself! So.. live demonstration: seach "ns-3 Time class" on Google

Example of scheduling

Static functions (C-style)

```
static void MyFn ()
{
    // ... do something
}

int main ()
{
    // ... initialize

    Simulator::Schedule (Seconds(5.0), &MyFn);
}
```

Example of scheduling

Classes (C++-style)

```
class MyClass
{
    void MyMember ()
    {
        // ... do something
    }
}

int main ()
{
    // ... initialize

    MyClass myClass;

    Simulator::Schedule (Seconds(5.0), &MyClass::MyMember, &my
```

Time to code!

- Open your favorite text editor (+1 for vim)
- Point it to examples/tcp/tcp-bulk-send.cc
- Do you remember it ?

First exercise:

Write a static function which prints "Hello, World" to standard output. Schedule it at the 15th second of simulated time.

Command:

```
./waf - -run "tcp-bulk-send"
```

Solution

```
static void Hello ()
{
  std::cout « "Hello, World!" « std::endl;
}

...

Simulator::Schedule (Seconds(15.0), &print);

//
// Now, do the actual simulation.
//
NS_LOG_INFO ("Run Simulation.");

...
```

Two words on logging

- std::cout and std::err are c++ standard, you can use them
- but it is appreciated to use different level of printed output
- you don't need all the prints all times: just write them, and then select the detail level you need

Log levels

There are currently seven levels of log messages of increasing verbosity defined in the system: ERROR, WARN, DEBUG, INFO, FUNCTION, LOGIC, ALL

Components

Logging is selective. It means that you can enable different log levels on different component. Usually, a component is a class.

Two words on logging

How to use them

```
NS_LOG_[LEVEL] ("Your message");

...

NS_LOG_UNCOND ("This is a print. oneVar= " « oneVar);
```

UNCOND is the "jolly" level. Always printed (syntactic replacement of std::cout) If you want to enable all messages ranging from DEBUG level to lower levels (the lowest is ERROR), run your program with:

```
export 'NS_LOG=ClassToDebug=level_debug'
./waf --run "tcp-bulk-send"
```

Time to code!

Declare a component

Do you notice the macro NS_LOG_COMPONENT_DEFINE? It defines the component for the logging system.

Exercise:

Enable (and see into a terminal) the INFO messages of TcpBulkSendExample.

Time to code!

Solution:

```
...
NS_LOG_INFO("Hello world!");
...

export 'NS_LOG=TcpBulkSendExample=level_all'
./waf --run "tcp-bulk-send"
```

Attributes

In ns-3 simulations, there are two main configuration aspects:

- The simulation topology and how objects are connected.
- The values used by the models instantiated in the topology.

We will focus on the second item: how the many values in use in ns-3 are organized, documented, and modifiable by ns-3 users. The ns-3 attribute system is also the underpin of how traces and statistics are gathered in the simulator.

Objects and their attributes

Object in ns-3

Many ns-3 objects inherit from the Object base class. These objects have some additional properties that we exploit for organizing the system and improving the memory management of our objects.

TypeId

ns-3 classes that derive from class Object can include a metadata class called TypeId that records meta-information about the class, for use in the object aggregation and component manager systems:

- A unique string identifying the class.
- The base class of the subclass, within the metadata system.
- The set of accessible constructors in the subclass.
- A list of publicly accessible properties (“attributes”) of the class.

Example of TypeId

Let's view src/point-to-point/model/point-to-point-net-device.cc togheter:

```
TypeId
PointToPointNetDevice::GetTypeId (void)
{
  static TypeId tid =
     TypeId ("ns3::PointToPointNetDevice")
    .SetParent<NetDevice> ()
    .SetGroupName ("PointToPoint")
    .AddConstructor<PointToPointNetDevice> ()
```

Example of TypeId - Continued

```
.AddAttribute (
   "Mtu", "The MAC-level Maximum Transmission Unit",
   UintegerValue (DEFAULT_MTU),
   MakeUintegerAccessor (&PointToPointNetDevice::SetMtu,
                         &PointToPointNetDevice::GetMtu),
   MakeUintegerChecker<uint16_t> ())
.AddAttribute (
   "DataRate",
   "The default data rate for point to point links",
   DataRateValue (DataRate ("32768b/s")),
   MakeDataRateAccessor (&PointToPointNetDevice::m_bps),
   MakeDataRateChecker ())
```

Example of TypeId - Explained

The `SetParent<NetDevice> ()` call in the definition above is used in conjunction with our object aggregation mechanisms to allow safe up- and down-casting in inheritance trees during GetObject (). It also enables subclasses to inherit the Attributes of their parent class.
The `AddConstructor<PointToPointNetDevice> ()` call is used in conjunction with our abstract object factory mechanisms to allow us to construct C++ objects without forcing a user to know the concrete class of the object she is building.

Example of TypeId - Explained

The three calls to `AddAttribute ()` associate a given string with a strongly typed value in the class. Notice that you must provide a help string which may be displayed, for example, via command line processors. Each Attribute is associated with mechanisms for accessing the underlying member variable in the object (for example, `MakeUintegerAccessor ()` tells the generic Attribute code how to get to the node ID above). There are also "Checker" methods which are used to validate values against range limitations, such as maximum and minimum allowed values.

Creation of PointToPointNetDevice

When users want to create objects, they will usually call some form of CreateObject (),:

```
Ptr<PointToPointNetDevice> n;
...

n = CreateObject<PointToPointNetDevice> ();
```

Creation of PointToPointNetDevice

Or more abstractly, using an object factory, you can create an object without even knowing the concrete C++ type:

```
ObjectFactory factory;
const std::string typeId = "ns3::PointToPointNetDevice";
factory.SetTypeId (typeId);
Ptr<Object> node = factory.Create <Object> ();
```

Both of these methods result in fully initialized attributes being available in the resulting Object instances.
We next discuss how attributes (values associated with member variables or functions of the class) are plumbed into the above TypeId.

Changing attributes

First: Identify the attribute

Look into source code of the class, then see the attributes declared in its TypeId

Example for DropTailQueue:

```
.AddAttribute ("MaxPackets",
               "The maximum number of packets accepted by this DropTailQueue.",
               UintegerValue (100),
               MakeUintegerAccessor (&DropTailQueue::m_maxPackets),
               MakeUintegerChecker<uint32_t> ())
```

Changing attributes - Default, for all instances

Change the attribute for all instances in the simulation:

```
Config::SetDefault (path, value)
Config::SetDefault ("ns3::DropTailQueue::MaxPackets",
                    UintegerValue (80));
```

- The path is a string, composed by `ns3::ClassName::AttributeName`
- To know the type of the value, check the default value in the TypeId declaration

Changing attributes - For only one instance

Change the attribute for only one instance:

```
Ptr<Queue> q = CreateObject<DropTailQueue> ();
q->SetAttribute ("MaxPackets", UintegerValue (80));
```

There are many other ways:

- CreateObjectWithAttributes
- API in *Helpers (do you remember what happens with PointToPointHelper?)
- Config Namespace Path ("/NodeList/0/DeviceList/0/TxQueue/MaxPackets")

Time to code!

Exercise:

Modify the hello function in tcp-bulk-send, in order to change the datarate of the point to point link to 2 Mbit/s at the 15th second of simulated time

Hint

- Read PointToPointNetDevice documentation for attributes
- Check how to get a PointToPointNetDevice pointer somewhere (from node or from container)
- Pass the pointer as parameter for the hello function
- Do the change

Solution

TODO

Callbacks

Definition of callback

In computer programming, a callback is a piece of executable code that is passed as an argument to other code, which is expected to *call back* (execute) the argument at some convenient time. The invocation may be immediate as in a *synchronous* callback, or it might happen at a later time as in an *asynchronous* callback. In all cases, the intention is to specify a function or subroutine as an entity that is, depending on the language, more or less similar to a variable.
Programming languages support callbacks in different ways, often implementing them with subroutines, lambda expressions, blocks, or **function pointers**. (Wikipedia)

Callbacks

How they are related to ns-3?

Short answer: thanks to callbacks, coupling between classes is not *strict*: register the callbacks you need/want, and make the simulator do the hard work

Example of C-style callback

Consider a function:

```
static double
CbOne (double a, double b)
{
  std::cout « "invoke cbOne a=" « a « ", b=" « b « std::endl;
  return a;
}
```

Consider also the following main program snippet:

```
int main (int argc, char *argv[])
{
  // return type: double
  // first arg type: double
  // second arg type: double
  Callback<double, double, double> one;
}
```

Example of C-style callback

Did you notice any similarity?

```
static    double CbOne (double a, double b)
             ^             ^         ^
             |             |         |
             |             |         |
Callback<double,       double,   double> one;
```

You can only bind a function to a callback if they have the matching signature. The first template argument is the return type, and the additional template arguments are the types of the arguments of the function signature.

Example of C-style callback

Now, let's instantiate the callback:

```
// build callback instance which points to cbOne function
one = MakeCallback (&CbOne);
```

Then, later in the program, if the callback is needed, it can be used as follows:

```
NS_ASSERT (!one.IsNull ());

// invoke cbOne function through callback instance
double retOne;
retOne = one (10.0, 20.0);
```

Example of C++-style callback

In C++, we have classes. What we need to pass cbTwo method as a callback?

```
class MyCb {
public:
  int CbTwo (double a) {
      std::cout « "invoke cbTwo a=" « a « std::endl;
      return -5;
  }
};
```

Example of C++-style callback

Just pass the pointer to the object as second argument.

```
int main ()
{
  ...
  // return type: int
  // first arg type: double
  Callback<int, double> two;
  MyCb cb;
  // build callback instance which points to MyCb::cbTwo
  two = MakeCallback (&MyCb::CbTwo, &cb);
  ...
}
```

Other callback API

Null callbacks:

```
two = MakeNullCallback<int, double> ();
NS_ASSERT (two.IsNull ());
```

Bound callbacks:

```
static void DefaultSink (Ptr<PcapFileWrapper> file, Ptr<const Packet> p);
...
MakeBoundCallback (&DefaultSink, file);
```

We can call the callback with only the packet as argument!

```
m_promiscSnifferTrace (m_currentPkt);
```

Tracing system

Simulation output

The whole point of running an ns-3 simulation is to generate output for studies. You have two basic strategies to obtain output from ns-3: using generic pre-defined bulk output mechanisms and parsing their content to extract interesting information; or somehow developing an output mechanism that conveys exactly (and perhaps only) the information wanted.

Tracing system

Tracing system

Trace sources are entities that can signal events that happen in a simulation and provide access to interesting underlying data. For example, a trace source could indicate when a packet is received by a net device and provide access to the packet contents for interested trace sinks. A trace source might also indicate when an interesting state change happens in a model. For example, the congestion window of a TCP model is a prime candidate for a trace source. Every time the congestion window changes, the connected trace sinks are notified with the old and new value.

Identify traced values

- By looking the source code of the desidered TypeId. Find ".AddTraceSource"
- By looking the definition of the desidered class. Find "TracedValue..."
- By looking into Doxygen documentation of the desidered class

Example, from point-to-point-net-device.cc :

```
.AddTraceSource ("MacTx",
  "Trace source indicating a packet has arrived "
  "for transmission by this device",
  MakeTraceSourceAccessor (&PointToPointNetDevice::m_macTxTrac
  "ns3::Packet::TracedCallback")
```

Create the callback

A TracedValue is born to be connected..

... to a callback. To create it, you need to discover its signature (return type and parameters): it is described in the TypeId

```
.AddTraceSource ("MacTx",
  ‘‘Trace source indicating a packet has arrived’’
  ‘‘for transmission by this device’’,
  MakeTraceSourceAccessor (&PointToPointNetDevice::m_macTxTrac
  "ns3::Packet::TracedCallback")
```

Create the callback

The type is **"ns3::Packet::TracedCallback"**, which means TracedCallback in Packet class.
A rapid search on Doxygen shows:

```
typedef void(*  TracedCallback)
     (const Ptr< const Packet > packet)
```

Create the callback

Now we know the type, let's create our static callback:

```
static void
MyTraceCb (const Ptr< const Packet > pkt)
{
    ...
}
```

Connect the value to the callback

```
...

Ptr<PointToPointNetDevice> dev = ... ;

dev->TraceConnectWithoutContext ("MacTx",
                                 MakeCallback (&MyTraceCb));
...
```

Outcome

So, that's all?

There are three levels of interaction with the tracing system:

- Beginning users can easily control which objects are participating in tracing.
- Intermediate users can extend the tracing system to modify the output format generated or use existing trace sources in different ways, without modifying the core of the simulator.
- Advanced users can modify the simulator core to add new tracing sources and sinks.

Time to code!

Get your hands on tcp-bulk-send and:

Easy Exercise:

Display the congestion window evolution of the TCP socket

Medium Exercise:

Lower the data rate of the p2p link (e.g. 10Kbit/s), then display all the losses in the associated DropTailQueue

Intermediate Exercise:

Add a Traced value to the source code of BulkSendApplication which indicates when it has transmitted all the data. Then, connect a Callback which prints this information to the screen.

Time to code!

The End!

Send your solutions by mail. Working ones will be published, best ones will be included in the presentation for next year (with appropriate credit): natale.patriciello@unimore.it