An end-to-end tour of a simulation

Tom Henderson    Mathieu Lacage

WNS3, March 2nd 2009
Outline

1. How a simulation is built

2. Diving in: topology construction

3. Diving In: an End To End Tour of a Packet
Outline

1. How a simulation is built
2. Diving in: topology construction
3. Diving In: an End To End Tour of a Packet
There are two ways to interact with the ns-3 API:

- **Construct a simulation with the *Container* API:**
  - Apply the same operations on sets of objects
  - Easy to build topologies with repeating patterns

- **Construct a simulation with the *low-level* API:**
  - Instantiate every object separately, set its attributes, connect it to other objects.
  - Very flexible but potentially complex to use

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

- One csma link
- One wifi infrastructure network
- Two ip subnetworks
- One udp traffic generator
- One udp traffic receiver
- Global god ip routing
Fire up an editor and look at the code
The *Low-Level* Version

Fire up an editor and look at the code
Outline

1. How a simulation is built
2. Diving in: topology construction
3. Diving In: an End To End Tour of a Packet
Why are objects so complicated to create?

We do:

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

Why not:

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

Or:

```cpp
Node node0 = Node ();
```
Templates: the Nasty Brackets

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

It is hard in C++:

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

So, we use:

- Reference counting: track number of pointers to an object (Ref+Unref)
- Smart pointers: Ptr<> , Create<> and, CreateObject<> 
- Sometimes, explicit Dispose to break cycles
Why don’t we have a MobileNode?

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

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

- A circular singly linked-list
- AggregateObject is a constant-time operation
- GetObject is a $O(n)$ operation
- Aggregate contains only one object of each type
The ns-3 type system

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

All this information is stored in `ns3::TypeId`:
- The parent type.
- The name of the type.
- The list of attributes (their name, their type, etc.).
- The list of trace sources (their name, their type, etc.).
The ns-3 type system

It is not very complicated to use:

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

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

- Define the features of your object:

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

- call `NS_OBJECT_ENSURE_REGISTERED`
XXX: maybe add more details.
1. How a simulation is built

2. Diving in: topology construction

3. Diving In: an End To End Tour of a Packet
User writes:

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

Application::Start:

```cpp
m_startEvent = Simulator::Schedule (startTime,
   &Application::StartApplication, this);
```
User calls Simulator::Run:

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

Socket::CreateSocket:

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

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

OnOffApplication::SendPacket:

Ptr<Packet> packet = Create<Packet> (m_pktSize);
m_txTrace (packet);
m_socket->Send (packet);
...
m_sendEvent = Simulator::Schedule(nextTime,
   &OnOffApplication::SendPacket, this);
Summary: how applications access network stacks

How to use a new protocol Foo:

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

How to implement a new protocol Foo:

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

ns-3 packets contain a lot of information:

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

ns-3 packets are magic:

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

Buffer

Data Zero Area Size Used start Used Size

Virtual Zero Area

Count Size Initial Start Dirty Start Dirty Size

Unused Area Dirty Area Unused Area

Used

Buffer Data

Data Zero Area Size Used start Used Size

Virtual Zero Area

Used
UDP Transmission I

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

UdpHeader udpHeader;
...
udpHeader.SetDestinationPort (dport);
udpHeader.SetSourcePort (sport);
packet->AddHeader (udpHeader);
Ptr<Ipv4L3Protocol> ipv4 =
    m_node->GetObject<Ipv4L3Protocol> ();
ipv4->Send (packet, saddr, daddr, PROT_NUMBER);
IPv4 Transmission I

Ipv4L3Protocol::Send:

Ipv4Header ipHeader;
...
ipHeader.SetSource (source);
ipHeader.SetDestination (destination);
ipHeader.SetProtocol (protocol);
ipHeader.SetPayloadSize (packet->GetSize ());
...
ipHeader.SetTtl (....);
...
Lookup (ipHeader, packet,
    MakeCallback (&Ipv4L3Protocol::SendRealOut, this));
Ipv4L3Protocol::Lookup searches a protocol which has an outgoing route for the packet and calls Ipv4L3Protocol::SendRealOut:

```cpp
packet->AddHeader (ipHeader);
Ptr<Ipv4Interface> outInterface =
    GetInterface (route.GetInterface ());
outInterface->Send (packet, ipHeader.GetDestination ()
Down, in ArpIpv4Interface:

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

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

```cpp
void
MyHeader::Serialize (Buffer::Iterator start) const
{
    start.WriteHtonU16 (m_data);
}

uint32_t
MyHeader::Deserialize (Buffer::Iterator start)
{
    m_data = start.ReadNtohU16 ();
    return 2;
}
```
Note: How Do you Implement a new Header? III

What really matters:

- Copy/Paste the code for GetTypeId and GetInstanceTypeId
- Make sure GetSerializedSize returns enough for Serialize
- Make sure Hton are balanced with Ntoh
- Remember that what is written in Buffer::Iterator must be faithful the the real network representation of the protocol header
ArpL3Protocol::Lookup:
  ● Try to find a matching live entry
  ● If needed, send an ARP request on NetDevice::Send
  ● Wait for reply

ArpL3Protocol::Receive:
  ● If request for us, send reply
  ● If reply, check if request pending, update cache entry, flush packets from cache entry
CsmaNetDevice Transmission

CsmaNetDevice::Send:

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

CsmaChannel::TransmitEnd:

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

CsmaNetDevice::Receive:

- Remove ethernet header and trailer
- Filter unwanted packets
- Apply packet error model
- Call device receive callback
During topology setup:

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

At runtime:

- Device calls `receive` callback to send packet to layer 3
- `Node::NonPromiscReceiveFromDevice` searches matching protocol handlers by protocol number
IPv4L3Protocol::Receive:

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

`WifiNetDevice::Send` is fairly simple:

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

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

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

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

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

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

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

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

- MacLow::CtsTimeout and MacLow::NormalAckTimeout notify upper layers
From the perspective of layer 2, it is a black box whose content is the topic of some presentations this afternoon!
MacLow::ReceiveOk handles incoming packets:

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

And notifies upper layers with its *receive* callback
MacRxMiddle::Receive:

```cpp
if (IsDuplicate (hdr, originator))
    return;
Ptr<Packet> agregate = HandleFragments (packet, hdr, originator);
if (agregate == 0)
    return;
m_callback (agregate, hdr);
```
Wifi Reception: MacHigh

- NqstaWifiMac::Receive:

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

- WifiNetDevice::ForwardUp: call the device receive callback
Summary: From Layer 3 to Layer 4

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

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

Ipv4EndPoint::ForwardUp calls into UdpSocketImpl::ForwardUp
```
```cpp
UdpSocketImpl::ForwardUp

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

PacketSink::HandleRead:

packet = socket->RecvFrom (from)
```
UdpSocketImpl::Recv

if (m_deliveryQueue.empty() )
{
    m_errno = ERROR_AGAIN;
    return 0;
}

Ptr<Packet> p = m_deliveryQueue.front();
if (p->GetSize () <= maxSize)
{
    m_deliveryQueue.pop ();
    m_rxAvailable -= p->GetSize ();
}

return p;