Network Simulator ns-3

Emanuele Giona Department of Computer Science, Sapienza University of Rome

Internet of Things A.Y. 2022/23

Prof. Chiara Petrioli Department of Computer, Control and Management Engineering, Sapienza University of Rome
Dott. Michele Nati

What is a network simulator?

It is a software program replicating the behavior of a real computer network.

This requires simulating:

- Nodes
- Links
- Routers
- Packets
- ...

and interactions among them.

Most importantly, simulators allow a deep parametric tuning of the network, allowing a wide range of supported configurations as well as environmental settings.

What is a network simulator?

Network simulators are useful:

- For educational purposes
  Topics related to computer networking, *i.e.* fundamental concepts, protocols, …
- For carrying out research
  Both in academia and industry

for a simple reason: inexpensive approach to fast iteration over ideas in order to reach a proof of concept.

What is a network simulator?

Main features of a network simulator:

- Definition of a network topology
  Set of nodes and links connecting them
- Model the application flow (traffic) in the network
- Allow for the computation of network performance metrics
  e.g. Packet Delivery Ratio, Latency, …
- Logging at multiple levels
  e.g. Packets, events, …

Some simulators also provide features to visualize the packet flow, but the main purpose of a simulator is to evaluate a technology stack or protocol design choices.

Technology stacks in simulators

Protocols are somewhat straightforward to simulate: they already are software.

Network simulators however need to provide a sufficient level of detail regarding the underlying network hardware too, in order to be particularly useful:

- Network interfaces
- Antennas
- Modulations to analog signals
- Channel models
- ...

A network simulation is as much as accurate as hardware models are close to real ones. A different usage may instead be modeling new hardware products in a simulator as to validate a proposal w.r.t. existing solutions.

Approaches to computer-based simulation

A key aspect in network simulation: time modeling.

- **Discrete-event simulation**
  Simulation consists of a discrete sequence of steps in time; each step is associated with a specific point in time.
- **Continuous simulation**
  Simulation consists of several variables that change their values according to a set of differential equations.

In particular, simulations are classified into either class w.r.t. states of the simulation.

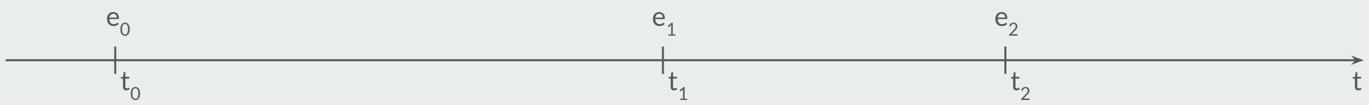
In continuous simulation variables are ever-changing due to the continuous updates; on the other hand, in discrete-event simulation variables are assumed to stay constant in between two steps, only updating during a step.

Approaches to computer-based simulation

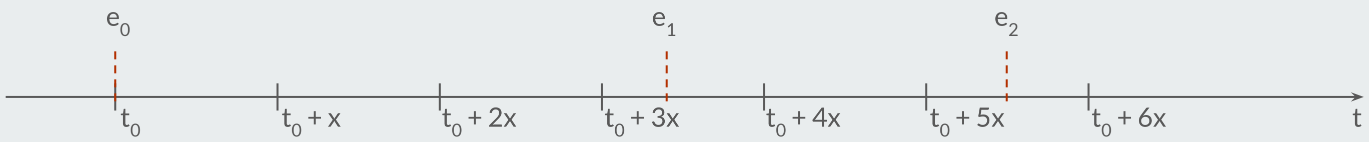
Network simulation is mostly implemented as a discrete-event simulation, also recalling how typical network stacks are implemented (*hint:* interrupts).

Discrete-event simulation can be implemented in two ways:

- **Next-event time progression**
  Each step represents a particular event in the network. Simulation is executed jumping from an event to the next.

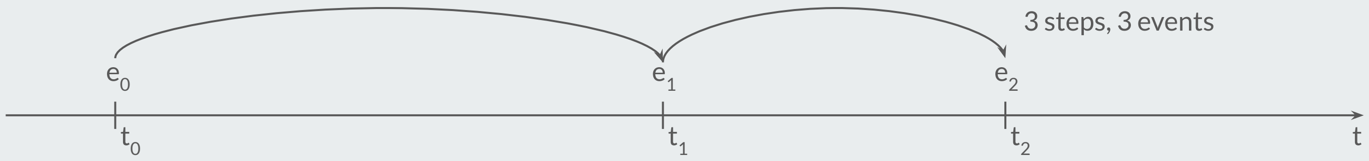


- **Fixed-increment time progression**
  Each step represents an equal-size time interval passed on the simulation timeline. Simulation is executed jumping from a time interval to the next.

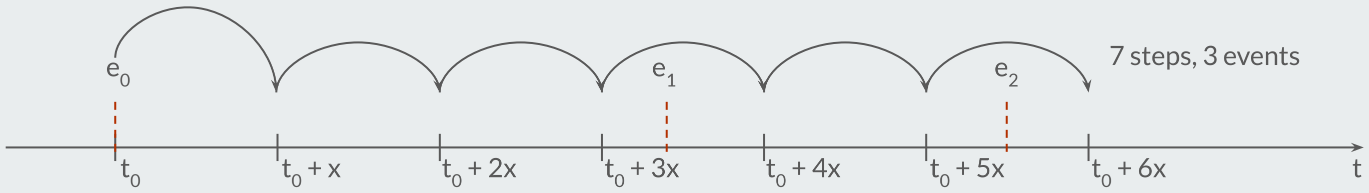


Approaches to computer-based simulation

- **Next-event time progression**
  Each step represents a particular event in the network. Simulation is executed jumping from an event to the next.



- **Fixed-increment time progression**
  Each step represents an equal-size time interval passed on the simulation timeline. Simulation is executed jumping from a time interval to the next.



Approaches to computer-based simulation

Fixed-increment time progression allows for more fine-grained tracking of system states, however after any interval the system state may be exactly equal to the one at the previous step, if no state-changing event occurred in the meantime.

As a matter of fact, next-event time progression is the most common implementation of discrete-event network simulators: this is due to a great reduction in execution times and the assumption that computer networks can be modeled well by an event-based system.

2. ns-3

Network simulator ns-3

ns-3 is a discrete-event simulator implementing next-event time progression.

- ➢ Open source project
- ➢ Developed in C++
- ➢ Supports simulation program drivers in C++ or Python
- ➢ No general retrocompatibility with ns-2
- ➢ Wide ecosystem of modules and tools

Source code download & installation instructions

It is recommended the use of a Linux-based system.

Docker images & usage instructions

Pre-compiled ns-3. Can run on any OS through Docker.

Network simulator ns-3

Modules

- Can only be written in C++
- Implement technology stack components or protocols
- Generally developed by researchers in academia or industry

Examples:

- Internet stack (TCP/IP)
- IEEE 802.11 (WiFi) MAC protocol
- AODV routing protocol

Simulation program drivers

- Can be written in either C++ or Python
- Implement a specific simulation scenario, complete of network topology and other settings
- Generally developed by anyone practicing with networking topics

Examples:

- Point-to-point network consisting of 2 nodes
- 5 nodes connected through WiFi at 1 access point
- Comparison of different TCP window sizes in an Internet network of 10 nodes with static routing

Point-to-point network in ns-3

This **simulation program driver** sets up a network consisting of 2 nodes communicating through a point-to-point link, using ns-3's C++ API.

Relevant libraries and ns-3 namespace

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");

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

  Time::SetResolution (Time::NS);
  LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
  LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
```

Logging utilities

The goal of simulation program drivers is to use existing modules and configure a specific simulation scenario.

In this case, TCP/IPv4 is going to be used and the two nodes are going to act as echo client and server.

Point-to-point network in ns-3

`NodeContainer` class

Represents a ns-3 container of objects of class Node. Such objects model a component of the network.

`PointToPointHelper` class

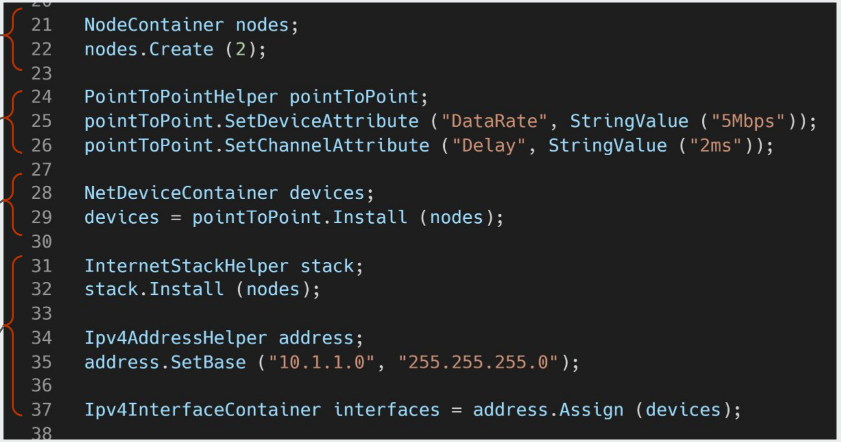
Utility class used to shorten configuration of PP links.

`NetDeviceContainer` class

Similar to `NodeContainer`, but with `NetDevice` objects.
Such objects model network interfaces of a Node.

TCP/IPv4 setup

Initialization (line 32), static IP configuration (line 35).



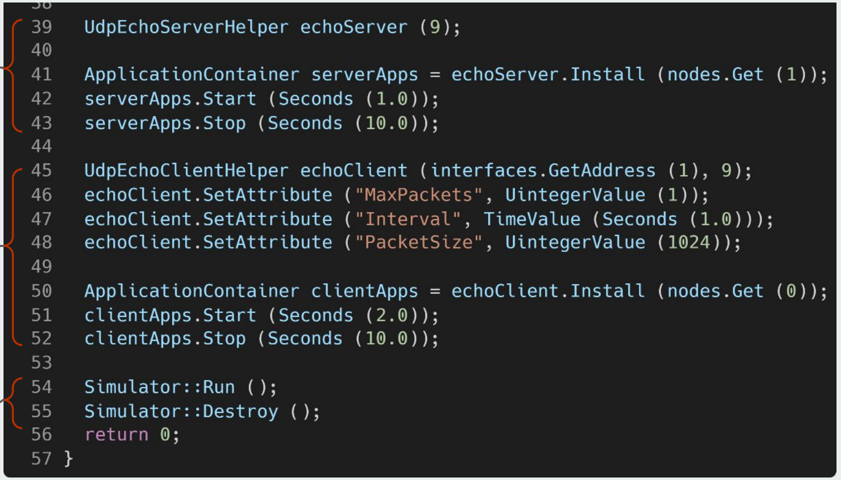
```
21   NodeContainer nodes;
22   nodes.Create (2);
23
24   PointToPointHelper pointToPoint;
25   pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
26   pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
27
28   NetDeviceContainer devices;
29   devices = pointToPoint.Install (nodes);
30
31   InternetStackHelper stack;
32   stack.Install (nodes);
33
34   Ipv4AddressHelper address;
35   address.SetBase ("10.1.1.0", "255.255.255.0");
36
37   Ipv4InterfaceContainer interfaces = address.Assign (devices);
38
```

Point-to-point network in ns-3

Echo server setup
UDP port 9 is going to be used (line 39), server installed on the second node (line 41); every packet between absolute timestamps 1 s ~ 10 s is echoed (lines 42-43).

Echo client setup
Destination set to the second node's interface with UDP port 9 (line 45), sending a single packet of size 1024 B every 1 s (lines 46~48); client installed on first node (line 50), working between absolute timestamps 2 s ~ 10 s (lines 51-52).

Simulation execution
The simulation starts at the execution of Simulator::Run(), returning only once events are not generated anymore; if Simulator::Stop() is used instead, execution will stop at the chosen time. Simulator::Destroy() clears memory used during execution (only internally!).



```
39   UdpEchoServerHelper echoServer (9);
40
41   ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
42   serverApps.Start (Seconds (1.0));
43   serverApps.Stop (Seconds (10.0));
44
45   UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
46   echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
47   echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
48   echoClient.SetAttribute ("PacketSize", UintegerValue (1024));
49
50   ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
51   clientApps.Start (Seconds (2.0));
52   clientApps.Stop (Seconds (10.0));
53
54   Simulator::Run ();
55   Simulator::Destroy ();
56   return 0;
57 }
```

First thoughts on ns-3 program drivers

In C++, there are multiple data structures and memory-related utilities provided within the STL.

During the first example however, `NodeContainer` and `NetDeviceContainer` classes are encountered: these are ns-3's specialized versions of containers. This will not be the only case in which *custom implementations* are used rather than building on top of STL.

This is because development of ns-3 started before the C++11 standard and also to attract users coming from ns-2, by keeping a few similarities with it. Most of the time, features of these objects are equal to what would have been implemented using STL.

Helper classes instead are a good practice when it comes to developing new modules for ns-3: most of a lengthy configuration is hidden from the simulation driver program. This prevents possible configuration errors from end users as well as allowing to focus on the specification of complex simulation scenarios in fewer lines of code.

Fundamental components and concepts

Node class

Objects of class Node represent a computing device in a network, on top of which one or more protocol stacks and applications can be installed.

```
NodeContainer nodes;
nodes.Create(5);
```

```
Ptr<Node> node = CreateObject<Node>();
```

Ptr: ns-3's smart pointer

CreateObject: Handles creation of a smart referenced object, given a type

While you can instantiate Node objects directly, relying on a `NodeContainer` object is more common. As previously mentioned, this special container handles the underlying instantiation and simplifies usage of ns-3's API.

This class does not have particular features built into it: easing up customization as well as real node usage.

`Object` class

In ns-3, Node is a derived class of `Object`: this is the base class for the most interesting classes.

Fundamental components and concepts

`Channel` class

Objects of this class represent the communication medium used to transmit to and received from. Upon registration on the same channel, `Node` instances will communicate with each other during the simulation. Several channel models are provided (*e.g.* Point-to-point), each implementing different characteristics.

A channel's behavior is greatly influenced by two features: propagation delay and propagation loss. Various models of both are already provided in ns-3 (*e.g.* Constant / Random delay; Friis / LogDistance / ecc loss), with possibilities of using libraries or custom implementations.

`NetDevice` class

Objects of this class represent a network interface of a `Node`: a `NetDevice` interacts with a `Channel` following a MAC protocol (*e.g.* Aloha, CSMA) as well as PHY-level settings (*e.g.* modulation, transmission power).

Similarly with `Node`, the `NetDevice` class has a limited API to allow the usage of real network interfaces installed on a computing device.

Fundamental components and concepts

`MobilityModel` class

Objects of this class represent the spatial behavior or a node: ns-3 allows the definition of static topologies as well as ones with moving nodes. Mobility of nodes is crucial when computing propagation delays and loss, and ns-3 provides many models for it (*e.g.* Constant position / Random walk in 2D / Waypoint / ecc).

Using ns-3 and launching simulations

After downloading and building the simulator, within the `ns-3.xx` directory you should find:

- `waf` / `ns3` executable
  It provides a wrapper for configuring build profiles (*e.g.* debug or optimized), executing simulation driver programs, running a driver program through valgrind and gdb tools, and more
- `scratch` directory
  This directory usually contains simulation driver programs
- `src` directory
  This directory contains source code for core and extension modules of ns-3; it should not be interacted with, unless a third-party module has to be manually installed
- `contrib` directory
  This directory usually contains custom modules that are still under development
- `utils` directory
  This directory contains utility scripts such as `create-module.py`, which is used to set up a new module's directory structure

You can run a simulation driver program stored in the scratch directory with the following command:

```
./waf --run scratch/file-name.cc
```

Logging utilities

ns-3 includes logging utilities, but they are only compiled when using the debug build profile. Once ns-3 is running in this profile, all enabled logging components will be able to output messages.

In order to define new logging components, there is a specific C++ macro to be used:

```
NS_LOG_COMPONENT_DEFINE("CustomModule");
```

This macro must be placed outside any function and it is generally positioned among the first lines of an implementation file (.cc).

Whenever there is the need to produce a logging message, a logging macro is defined for each severity level:

LOG_NONE	-
LOG_ERROR	NS_LOG_ERROR
LOG_WARN	NS_LOG_WARN
LOG_DEBUG	NS_LOG_DEBUG
LOG_INFO	NS_LOG_INFO
LOG_FUNCTION	NS_LOG_FUNCTION
LOG_LOGIC	NS_LOG_LOGIC

When enabling a logging component, is it possible to turn on logging for a specific severity level and all others above it.

Examples:

- `LogComponentEnable("CustomModule", LOG_LEVEL_ERROR);` will only show output from `LOG_ERROR` level
- `LogComponentEnable("CustomModule", LOG_LEVEL_INFO);` will show output from `LOG_INFO`, `LOG_DEBUG`, `LOG_WARN`, and `LOG_ERROR` levels

3. Keystones of ns-3

Smart pointers

ns-3 makes use of an own implementation of smart pointers `Ptr<T>`, instead of relying on the ones from C++'s STL. Their features are equivalent, but their usage is simplified: any class deriving from `SimpleRefCount` (included by ns-3) is allowed to be used with the template class `Ptr`.

The main advantage of using a smart reference instead of raw pointers consists in automatic memory management. As a result, ns-3 widely employs them: `NodeContainer` and `NetDeviceContainer` classes indeed only make available `Ptr<Node>` and `Ptr<NetDevice>` objects, respectively.

Access to the referenced objects occurs in the same way as it would be with raw pointers: through the arrow operator (`->`). Assigning `0` to a smart pointer object will deallocate the referenced object, if no other references are currently in use, or decrease the internal references counter.

```
Ptr<Node>                                                n;
…
if(n)
  n = 0
```

Checking pointers' validity before operating on them

Callbacks

A callback function is a reference to executable code that is passed as argument to other code. The other code is expected to *call back* (execute) such code at a later time.

C++ allows to specify pointers to functions, treating them in the same way as other pointers. This means that we may have function pointer variables, arrays, ecc.

```
callback fn = printToStdout;
```

Variable type (named function pointer via `typedef`)

Function name (*i.e.* pointer to the memory location of the entry point to the executable code)

```
#include <iostream>
#include <string>
using namespace std;

void printToStdout(string str){
  cout << str << endl;
}

typedef void (*callback)(string);

int main()
{
  callback fn = printToStdout;
  fn("Hello World!");
  return 0;
}
```

This is actually a C-style function pointer. C++ provides other mechanisms too (*e.g.* Callables), but they are out of our scope.

Callbacks in ns-3

ns-3 provides a standardized implementation of callbacks that can be easily accessed by including the appropriate header file (`ns3/callback.h`).

```
Callback<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9> fn;
```

Mandatory, represents return type

Optional, represent types for up to 9 arguments

In order to instantiate the variable `fn`, ns-3 provides two mechanisms:

- ➤ `Callback<void, string> fn = MakeCallback(&printToStdout);`
  All callback arguments must be passed upon callback invocation (*e.g.* `fn(“Hello World!”)`)
- ➤ `Callback<void> hwFn = MakeBoundCallback(&printToStdout, “Hello World!”);`
  It allows to provide a fixed argument (the first one), but all remaining arguments must be passed upon callback invocation (*e.g.* `hwFn();` but there are no additional arguments here); notice the variable type: the callback is considered to have no arguments

Callbacks using member functions

ns-3 also allows for the use of member functions in callbacks. In this case, additional care must be taken, providing the function's full qualifier (`ClassName::FunctionName`) as well as a pointer to an instance of the class containing such function.

```cpp
#include <iostream>
#include "ns3/core-module.h"
#include "ns3/callback.h"
using namespace ns3;

class Example
{
public:
  void PrintToStdout(std::string str)
  {
    std::cout << str << std::endl;
  }
};

int main(int argc, char *argv[])
{
  Example ex;
  Callback<void, std::string> fn = MakeCallback(&Example::PrintToStdout, &ex);
  fn("Hello World!");
  return 0;
}
```

`PrintToStdout` defined as member function of class `Example`

Instance of class `Example`

`PrintToStdout`'s full qualifier; additionally, the first argument after the function must be the pointer to the class instance to be used

Perks of deriving from `Object` class: aggregation

As previously mentioned, Node is a derived class of `Object` in ns-3. This entails inheriting the usage through smart pointers (`Ptr<Node>`) as well as other powerful features: object aggregation and access to the attribute system.

Object aggregation is the reason why there is a single Node class and there is no need to extend it when installing a specific protocol stack.

```
Ptr<Ipv4L3Protocol> ipv4 = CreateObject<Ipv4L3Protocol>();
ipv4->SetNode(node);
node->AggregateObject(ipv4);
Ptr<Ipv4Impl> ipv4Impl = CreateObject<Ipv4Impl>();
ipv4Impl->SetIpv4(ipv4);
node->AggregateObject(ipv4Impl);
```

Through `Object::AggregateObject`, you can attach an instance of a class derived from `Object`.
Only a single instance of a given class can be attached at a time: using aggregation on a second `Ipv4Impl` is not supported.

```
Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4>();
```

`Object::GetObject<DesiredClass>` can be used to retrieve the single instance of class `DesiredClass` attached. If the instance is not present, the returned pointer will not be valid!

This mechanism allows to define complex interactions among classes, avoiding the need to define custom derived classes every time. Furthermore, instances of `Object`-derived classes can be created through `ObjectFactory`.

Perks of deriving from `Object` class: attributes

When deriving from `Object`, each class must implement a few functions in order to successfully inherit from `Object`. One of them is `GetTypeId`: this member function is used to set up class metadata within ns-3's hierarchy, including the accepted attributes.

```cpp
TypeId
Example::GetTypeId(){
  static TypeId tid = TypeId("ns3::Example")
    .SetParent<Object>()
    .AddConstructor<Example>()
    .AddAttribute("Attr", "Attribute example",
                  IntegerValue(0),
                  MakeIntegerAccessor(&Example::m_attr),
                  MakeIntegerChecker<int>());
  return tid;
};
```

```cpp
int main(int argc, char *argv[])
{
  Ptr<Example> ex = CreateObject<Example>();
  std::cout << ex->GetAttr() << std::endl;
  ex->SetAttribute("Attr", IntegerValue(5));
  std::cout << ex->GetAttr() << std::endl;
  return 0;
}
```

This mechanism provides a standardized way of accessing any component's tunable parameters in several convenient ways, also handling default values and supported value ranges. It can be combined with the `ObjectFactory` instantiation, obtaining a very flexible tool for creating Object-derived classes' instances.

Packets

As stated before, packets are crucial for the usefulness of network simulator. Considering how many packets can be exchanged within a network for any given interval, it is likewise important to handle them in an efficient manner.

In particular, ns-3's `Packet` implementation takes into account the following issues:

- Consistency with core parts of the simulator
- Fragmentation & concatenation features
- Memory efficiency
- Support for dummy data (*e.g.* synthetic applications)

Packets

The issue with consistency with ns-3's core arises when defining new specialized packets for protocols.

The solution relies on the a similar mechanism to object aggregation: the `Packet` class is not extended, but custom headers, trailers, and tags are attached to its instances. Classes deriving from `Header`, `Trailer`, and `Tag` are used to implement the desired packet structure and other features; serialization/deserialization operations must be implemented explicitly.

Each `Packet` consists of:

- Byte buffer
  Serialized content, it should be identical to what is found in a real packet
- List of byte `Tags`
  Tags associated with specific bytes in the buffer; they will only be present in the fragment containing those specific bytes
- List of packet `Tags`
  Tags associated with the whole packet; they will be present on all fragments of the packet
- `PacketMetadata` object
  Utility functions (*e.g.* enabling printing, checking, adding/removing headers/trailers)

Packets

The issue with memory efficiency arises due to the great amount of packets typically shared during a simulation, no matter how short it can be.

The solution mainly relies on:

- Widespread usage of smart pointers, but in a more lightweight fashion that typically w.r.t. Object
- Byte buffer does not allocate any memory when using dummy data
- Copy-on-Write technique

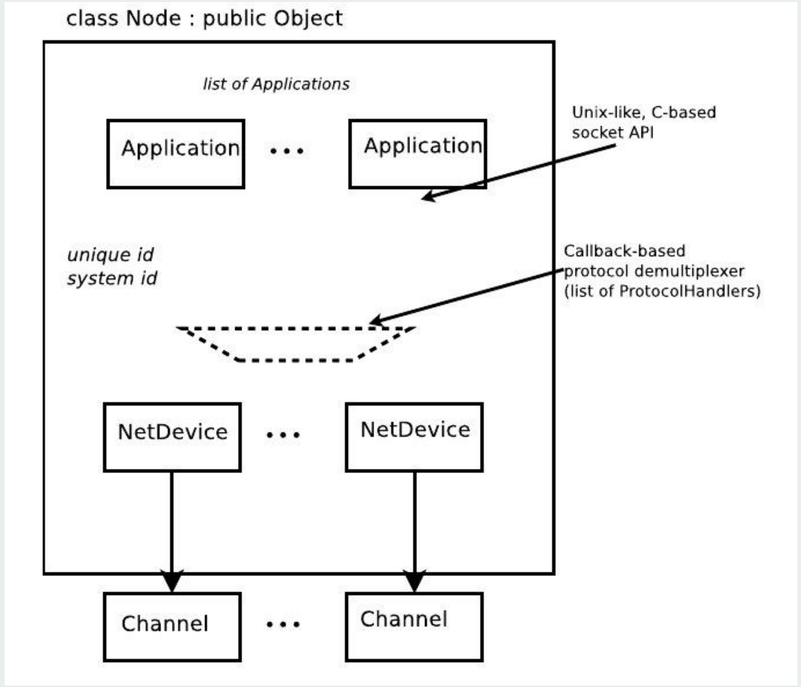
  Only few operations trigger the copy of a packet: `AddHeader`, `RemoveHeader`, `AddTrailer`, `RemoveTrailer`, `AddPacketTag`, `RemovePacketTag`, `AddAtEnd` (concatenation); all other operations use the same packet through the pointer

4. Protocol stack execution

Protocol stack

In ns-3 there is very much freedom to implement the protocol stack.

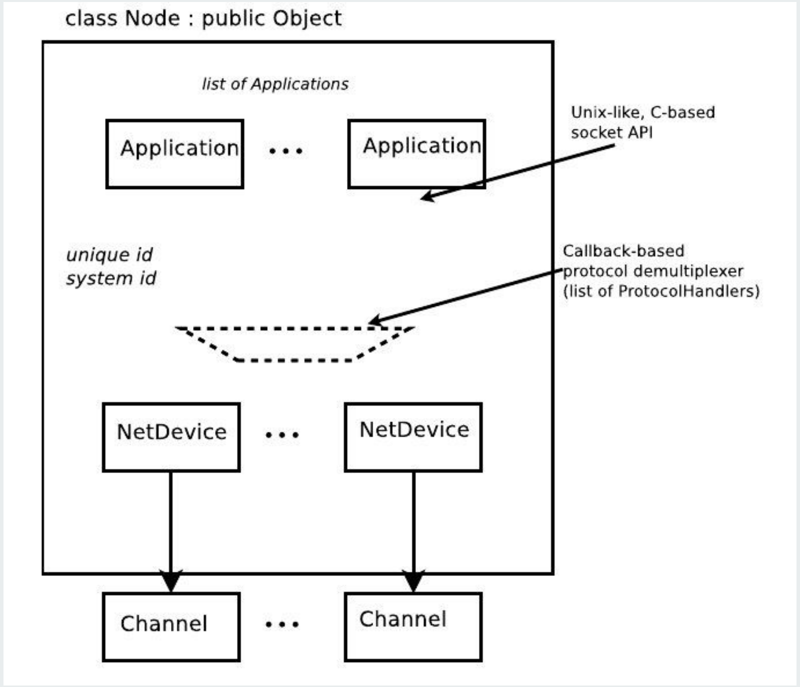
- ➢ PHY layer can be configured with custom modulations, SINR models, PER models, and transducers
- ➢ MAC layer can be configured with custom protocols; this layer is encapsulated within `NetDevice` instances
- ➢ APP layer can rely on the provided applications built-in with ns-3, or custom applications can be implemented


class Node : public Object
list of Applications
Application
Application
Unix-like, C-based socket API
unique id
system id
Callback-based protocol demultiplexer (list of ProtocolHandlers)
NetDevice
NetDevice
Channel
Channel

Protocol stack

Anything in between APP and MAC layers can be freely implemented with no specific constraints. However, ns-3 also implements a Socket API that completely resembles BSD sockets, and it can be used to interconnect the layers within a protocol stack.

A protocol demultiplexer consults a list of `ProtocolHandlers`, each registering callbacks exposed by upper-layer protocols to events of their interest (*e.g.* reception of a packet at MAC layer). Upon the occurrence of a registered event, the protocol demultiplexer takes care of invoking the callback associated with it, effectively linking a lower and an upper layer.


class Node : public Object
list of Applications
Application
Application
Unix-like, C-based socket API
unique id
system id
Callback-based protocol demultiplexer (list of ProtocolHandlers)
NetDevice
NetDevice
Channel
Channel

Sending a packet: a journey from APP to PHY

1. APP initiates packet transmission via `Socket::Send()`
2. Implementation-specific `Socket` instance:
   a. Retrieves “lower layer” object (usually via aggregation)
   b. Invokes appropriate functions of that layer’s protocol, passing the packet
   c. *(Repeat from point 2a until all intermediate layers are finished)*
   d. Proceed to point 3
3. `NetDevice` passes the packet to its embedded MAC layer
4. MAC layer interacts with PHY according to the MAC protocol, ultimately sending the packet over the `Channel` instance it is associated with
5. `Channel` receives the packet and copies of it are delivered to other node’s PHY layers after a delay (computed based on configurable channel’s logic)

Receiving a packet: a journey from PHY to APP

1. Upon successful reception of a packet, PHY layer invokes a callback to notify this event
2. The MAC protocol function which served as callback at point 1 is executed: according to its logic, if the packet is intended for the upper layer, a callback is invoked to notify this event
3. The protocol demultiplexer is invoked, matching the event of receiving a packet with the registered callback on the basis of protocol number and `NetDevice` instance producing the event; in general, the callback is a function of an implementation-specific `Socket` instance
   a. `Socket` retrieves "upper layer" object (usually via aggregation)
   b. Invokes appropriate functions of that layer's protocol, passing the packet
   c. *(Repeat from point 3a until all intermediate layers are finished)*
   d. Proceed to point 4
4. `Socket` invokes callback to handle outgoing packets
5. APP layer's function which served as callback at point 4 is executed: the packet has been successfully received by the application

References

- https://en.wikipedia.org/wiki/Network_simulation
- https://en.wikipedia.org/wiki/Discrete-event_simulation
- https://en.wikipedia.org/wiki/Continuous_simulation
- https://www.nsnam.org/
- https://gitlab.com/nsnam/ns-3-dev
- https://github.com/SENSES-Lab-Sapienza/ns3-woss-docker
- https://gitlab.com/nsnam/ns-3-dev/-/blob/master/examples/tutorial/first.cc
- https://gitlab.com/nsnam/ns-3-dev/-/blob/master/examples/tutorial/first.py
- https://www.nsnam.org/docs/tutorial/html/
- https://en.cppreference.com/w/cpp/memory/unique_ptr
- https://en.cppreference.com/w/cpp/memory/shared_ptr
- https://en.wikipedia.org/wiki/Callback_(computer_programming)
- https://en.cppreference.com/w/cpp/named_req/Callable
- https://www.nsnam.org/docs/manual/html/callbacks.html
- https://www.nsnam.org/docs/release/3.35/manual/html/object-model.html
- https://www.nsnam.org/docs/release/3.35/manual/html/attributes.html
- https://www.nsnam.org/docs/release/3.35/models/html/packets.html