Middleware Lab 01 : Introduction to OMNeT++

Guillaume-Jean Herbiet
<guillaume.herbiet@uni.lu>

October 31, 2008

Abstract

This first lab aims at discovering event-based discrete network simulation and the usage of the OMNeT++ network simulator.

During this lab, we will start from the creation of a simple wired network and learn how to use the OMNeT++ simulator while improving this model by adding features to the hosts behavior, collecting and displaying statistics and learning how to deal with simulation parameters.

At the end of this lab, you will have modeled a simple wireless network, using IEEE 802.11 (Wi-Fi) for communication and providing a simple broadcast-based application.

1 Getting started

In this section, you will get familiar with the chosen environment for modeling, development and simulation.

1.1 Operating system and environment

For this project, we will use OMNeT++ version 3.4b2 and the Mobility Framework version 2.0p3 under GNU/Linux (Ubuntu 8.04.1 as distribution).

For sake of convenience, you are provided a VMWare appliance of this operating system with OMNeT++ and Mobility Framework already installed from source with proper environment customization. Those who are interested on how to install OMNeT++ may refer to the installation section of the user manual and the `README` file located in `/home/user/Development/sources/`.

Username and password to login are simply `user`.

OMNeT++ has been installed in the subdirectory `Development` of your home folder, so it is available at `/home/user/Development/omnetpp-3.4b2`

1.2 Development process

OMNeT++ generates simulations as UNIX executables from NED and C++ code and uses dedicated tools to generate makefiles for proper compilation. Therefore, it is neither required nor recommended to use an IDE for simulation development.

But, if you are familiar with Eclipse CDT or Anjuta, you can install them on your virtual machine (`user` is amongst the sudoers).

However, once you have described your custom modules using a NED file and implemented their functionality in C++ or defined messages in a MSG file, the simplest way is to use the commands provided by OMNeT++ in a Terminal window:

- `opp_msgc` will compile a message (MSG file into C++)
- `opp_makemake` will generate the makefile with all required dependencies for your project

For convenience, a script called `opp_mf_makemake` is provided and will generate a makefile for projects using the Mobility Framework .

Once your messages are compiled and proper makefile generated, simply type `make` to build your project.

1.3 Help and documentation

To help you during your development, the default web browser of the appliance contains links to the documentation provided with OMNeT++ and the Mobility Framework :

- OMNeT++ manual
- OMNeT++ API documentation
- TicToc tutorial (which served as basis for this first lab)
- Mobility Framework manual
- Mobility Framework API documentation

Besides, note that OMNeT++ and Mobility Framework provide a lot of implemented modules and templates for you to start with. These files are excellent examples for how to implement more complicated modules. See for instance `omnetpp-3.4b2/samples`, `omnetpp-3.4b2/mobility-fw2.0p3/networks` or `omnetpp-3.4b2/mobility-fw2.0p3/templates`

2 Creating a first network

In OMNeT++ , networks and nodes inside the network are described using the NED (NEtwork Description) language. In this first step, we will try to become familiar with this langage by setting up a first network that will be used and modified in all the upcoming lab sessions.

2.1 Objective

In this first step, we will build a two-node wired network. At simulation startup, the first node will send a message to the second node. Then, upon receiving, each node will re-send the message back to the sender.

2.2 Network topology

First, create a project directory called `lab01` as a subdirectory of `Development`

In your project directory, we will now describe what will be the basic topology of our simulated network. Therefore, create a file called `Network.ned` and open it for edition. Then insert the following code :

Listing 1: Network.ned

```
simple SimpleHost
    gates:
        in: in;
        out: out;
endsimple

module Sim
    submodules:
        host0: SimpleHost;
        host1: SimpleHost;
    connections:
        host0.out --> delay 100ms --> host1.in;
        host0.in <-- delay 100ms <-- host1.out;
endmodule

network sim : Sim
endnetwork
```

The description of a network in a NED file goes from the most particular to the most generic item. For proper understanding it is thus more convenient to explain its contents reading it backwards :

- The last paragraph creates the simulated network `sim` which is of module type `Sim`;
- The `Sim` module type is described in the second paragraph. It contains :
  - A list of modules contained in this module, introduced by the keyword `submodule`. The name of each submodule and its type are given;
  - The connections between each submodule are described.
- Finally, the first paragraph gives a description of the submodule types used in the Sim module type (i.e. `SimpleHost` here). As the model `SimpleHost` is kept simple, only the gates used for module connection have to be described.

Note that in OMNeT++ , links are half duplex so two connections have to be created so as to connect two hosts in a symmetrical way.

2.3 Model implementation

2.3.1 Compound modules and simple modules

In OMNeT++ , modules can be of two kind:

- **Compound module**, if the module contains other modules, as described in the NED file. This is the case for `Sim` for instance;

- **Simple module** when this module contains no other nested submodule, like `SimpleHost` in the previous example.

Compound modules are used for structuring the model and have no real purpose besides being placeholders for simple modules.

Simple modules are where modeling occurs. Simple modules require an associated C++ class to implement their behavior in reaction to some events.

As `SimpleHost` is a simple module, we will need to create an associated C++ class.

2.3.2 Implementing simple module behavior

In your project directory, create files respectively called `SimpleHost.h` and `SimpleHost.cc`. These will be used to implement the class associated with the `SimpleHost` module.

In the `SimpleHost.h` file, insert the following content :

Listing 2: SimpleHost.h

```
#ifndef SIMPLEHOST_H_
#define SIMPLEHOST_H_

#include <stdio.h>
#include <string.h>
#include <omnetpp.h>


class SimpleHost : public cSimpleModule
{
protected:
    // The following redefined virtual function holds the algorithm.
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

#endif /* SIMPLEHOST_H_ */
```

Note that OMNeT++ provides a very rich API that will take care of many low-level features (like event-list management, etc...). Practically, this induces the fact that our `SimpleHost` class extends the generic `cSimpleModule` class that provides a lot of helpful functions.

For this simple model, we only need to implement the basic functions each module should have[1]:

- `initialize();` that describes what has to be done when the module is created (i.e. just before the simulation starts);
- `handleMessage();` that tells what action to perform when a message is received.

Note that those two functions are `protected` and `virtual` as they are made to be redefined, would we like to create a class extending `SimpleHost`.

[1] We will see further that, especially when statistics collection is involved, the `finish()` function is also required.

Now, the `SimpleHost.cc` file will hold the actual implementation for those two functions:

Listing 3: SimpleHost.cc

```
#include "SimpleHost.h"

// The module class needs to be registered with OMNeT++
Define_Module(SimpleHost);

void SimpleHost::initialize()
{
    if (strcmp("host0", name()) == 0)
    {
        ev << "I'm host 0, sending initial message" << endl;
        cMessage *msg = new cMessage("SimpleMsg");
        send(msg, "out");
    }
}

void SimpleHost::handleMessage(cMessage *msg)
{
    ev << "Received message `" << msg->name() << "'" << endl;
    send(msg, "out");
}
```

Note the presence of the `Define_Module` statement which is very important. It should be present in all classes you implement as it makes the link between this class and the corresponding NED description of a module. If you forget one, the simulation will crash and let you know that some module misses this declaration.

In the `initialize()` function, we use the `name()` function, offered by OMNeT++ API, that returns the name of the module to decide which host should send the first message.

The message is created with the generic `cMessage` kind and sent on gate `out`, which was defined in the `Network.ned` file as being connected to the entering gate of the other module, using the `send()` function.

Finally, upon message reception, we simply print a message and send the message out again to the other module.

2.4 Simulation configuration, build and run

It is now time to parameter the simulation. This is done by creating the `omnetpp.ini` file into your project directory.

So far, this file can be kept very simple (but we will use it more intensively in the next steps):

Listing 4: omnetpp.ini

```
[General]
preload-ned-files = *.ned
network = sim
```

Here we simply say to load all NED files present in the directory (all modules created will be parsed, even if not actually used in the simulation), and declare that the default network to simulate is the network called `sim`.

Now build your simulation (using `opp_makemake` and `make`) and run the created executable (which should be named `lab01`).

OMNeT++ graphical environment (tkenv) should start, select `sim` as network to simulate and press "Run". You should see the two modules exchanging the message called `SimpleMessage`.

3 Limiting the number of message exchanges

3.1 Objectives

You may have noticed that the previous simulation can run forever as nothing will perturb the message exchange. In this step, we will implement a limitation of the number of transmissions a node can perform. This can, for instance, model the energy consumption of nodes running on batteries that will turn off after transmitting too many messages.

Doing this, we will introduce module parameterization and allow users to follow evolution of a module attribute during the simulation.

3.2 Adding transmission limit parameter to model

We want first to declare that SimpleHost has a parameter, which is the maximum number of transmission the module can do, so we need to modify `Network.ned` as follows:

Listing 5: Network.ned

```
simple SimpleHost
    parameters:
        limit: numeric const;
    gates:
        in: in;
        out: out;
endsimple
```

It is possible to declare values for this parameter either when we declare a module of kind `SimpleHost` in a compound module, or using the `omnetpp.ini` file.

We will use the first choice for `host0`, the second for `host1`, so the `Network.ned` and `omnetpp.ini` files have to be modified as such:

Listing 6: Network.ned

```
module Sim
    submodules:
        host0: SimpleHost;
            parameters:
                limit = 8;
        host1: SimpleHost;
        connections:
            host0.out --> delay 100ms --> host1.in;
```

```
9           host0.in <-- delay 100ms <-- host1.out;
endmodule
```

Listing 7: omnetpp.ini

```
[General]
preload-ned-files = *.ned
3 network = sim

[Parameters]
6 sim.host1.limit = 5
```

In the `omnetpp.ini` file, parameters are identified by their *full path*. For instance, here we have defined the parameter `limit` of the module `host1`, which is compound in the root module `sim`.

It is also possible to set the same parameter value to multiple modules using wild-cards. For instance `sim.host*` will address all submodules of `sim` whose name starts with `host`.

Note that it is more convenient to set values for all parameters in the `omnetpp.ini` as it allows to change it without re-compiling the model, and generate different simulation runs with varying parameters (as we will see later during this project)[2].

3.3 Implementing transmission limitation

In the `SimpleHost` model (i.e. the `.h` and `.cc` files), implement the following :

- Get the value of the *limit* parameter and store it into a variable called `limit` at module initialization (using the `par()` function);
- Create a `counter` variable and make it watchable during the simulation (using the `WATCH()` function);
- Upon message reception, check if the limit of transmission is reached, if not send back message and increment the number of transmission, otherwise delete message. In each case, generate text output (using `ev`) for users to follow what is happening.

3.4 Simulation build and run

Build this simulation (you can use `make clean` to delete objects from the previous compilation) and run it.

Click on the `host0` and `host1` objects. In the opened windows, in the *Contents* pane, look for the counter evolution.

This time, when `host1` hits its transmission limit, the simulation should stop as there is no more event to execute.

[2] In case where parameters are given a value in both places, the `omnetpp.ini` file has precedence over the `.ned` file

4 Towards more realistic network

4.1 Objectives

In this step, we will generate a network with more hosts (the number of hosts will be a simulation parameter). Therefore we will need to perform some kind of routing: each node will have to decide whether or not to forward a message and to who.

As a consequence, we will also design our own message type, so as to allow the routing operation.

4.2 Implementing a message type for routing

Now that we have more than two hosts, messages can require multiple relays to reach their final destination. It is therefore required that each message carries the address of its final destination. We will also add the address of the source and a counter of the number of hops required to reach the destination.

For this purpose, we will implement the `SimpleMessage` message class by creating a `SimpleMessage.msg` file with the following content:

Listing 8: SimpleMessage.msg

```
message SimpleMessage
{
    fields:
        int source;
        int destination;
        int hopCount = 0;
}
```

This creates a message with the `source`, `destination` and `hopCount` fields. The latter is set with default value 0.

This message file has to be compiled using the command `opp_msgc SimpleMessage.msg`.

This will automatically generate an associated class `SimpleMessage` (described in `SimpleMessage_m.h` and `SimpleMessage_m.cc`) with *setters* and *getters* to the defined fields.

For each field, `myField` two methods are created : `getMyField()` that will return the field current value and `setMyField()` that allows to set the field to the value passed as parameter.

4.3 Generating a n-node network

4.3.1 Number of hosts as a parameter

As we want `numHosts` to be a simulation parameter, it has to be declared as such in the `Network.ned` file:

Listing 9: Network.ned

```
network sim : Sim
    parameters:
        numHosts =input(5,"Number of hosts:");
endnetwork
```

The `input` keyword will trigger a message box asking you for *Number of hosts:* with default value of *5*. You can also try to define the corresponding parameter in the `omnetpp.ini` file.

4.3.2 Automatically generating a topology

Now that the number of hosts is dynamic, it is impossible to keep the current static declaration of nodes connections in the `Sim` definition.

Therefore, modify the `Sim` module definition as follows:

Listing 10: Network.ned

```
module Sim
    parameters:
        numHosts: numeric const;

    submodules:
        host: SimpleHost[numHosts];
        connections:
            host[numHosts-1].out --> delay 100 ms --> host[0].in;
            for i=0..numHosts-2 do
                host[i].out --> delay 100ms --> host[i+1].in;
            endfor;
endmodule
```

This description will generate a vector of *numHosts* modules of kind `SimpleHost` that will be named `host[0]` to `host[`*`numHosts-1`*`]`.

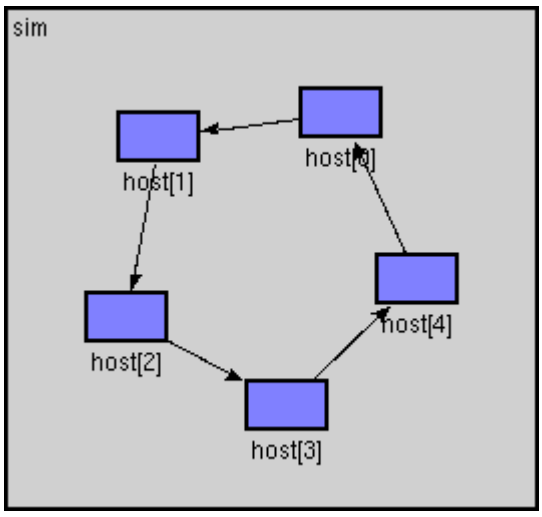

Figure 1: A simple ring topology with 5 hosts

The `for` loop will generate a simple ring. You can look at the networks examples in `omnetpp-3.4b2/samples` for more interesting topologies.

In those topologies, as nodes may have multiple `in` and `out` gates, they are considered as vectors of gates, and referred as `in[`*`index`*`]` and `out[`*`index`*`]`.

4.4 Performing simple routing

The `SimpleMessage` class allows to store the source and destination address of each message. This will be useful for routing operation.

Therefore we propose to modify the `SimpleHost` model by creating the following functions :

- `generateMessage()` that will create a `SimpleMessage` with host index as source address and random destination address. The host with index 0 will generate a `SimpleMessage` at initialization;
- `forwardMessage()` that will randomly select a next hop amongst neighbors (i.e. a random `out` gate in the gate vector) to forward the received message.

Each time a message is received, a host will forward it if it is not the final destination. Otherwise, it will delete it and generate a new message.

Those operations will require you to find the appropriate methods to get host index, size of out gates vector, generate random numbers. You can look to OMNeT++ manual and/or API documentation to find those.

So as to help you, here is the template for the modified `SimpleHost` class:

Listing 11: SimpleHost.h

```
#ifndef SIMPLEHOST_H_
#define SIMPLEHOST_H_

#include <stdio.h>
#include <string.h>
#include <omnetpp.h>

#include "SimpleMessage_m.h"

class SimpleHost : public cSimpleModule
{
private:
    int counter;
    int limit;

protected:
    // The following redefined virtual function holds the algorithm.
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);

    virtual SimpleMessage *generateMessage();
    virtual void forwardMessage(SimpleMessage *msg);
};

#endif /* SIMPLEHOST_H_ */
```

Note that the `SimpleMessage_m.h` header file is included to allow references to the `SimpleMessage` class.

Listing 12: SimpleHost.cc

```
#include "SimpleHost.h"
```

```
// The module class needs to be registered with OMNeT++
Define_Module(SimpleHost);

void SimpleHost::initialize()
{
    // Variables initialization here
    ...

    if (...)
    {
        ev << "I have index 0, sending initial message" << endl;
        SimpleMessage *msg = generateMessage();
        forwardMessage(msg);
    }
}

void SimpleHost::handleMessage(cMessage *msg)
{
    SimpleMessage *smsg = check_and_cast<SimpleMessage *>(msg);

    // I am final destination
    if (...)
    {
        ev << "Message " << smsg << " arrived after "
           << smsg->getHopCount() << " hops.\n";
        delete smsg;

        // Generate another one.
        ev << "Generating another message: ";
        SimpleMessage *newmsg = generateMessage();
        forwardMessage(newmsg);
    }
    else
    // I am not final destination
    {
        forwardMessage(smsg);
    }
}

SimpleMessage* SimpleHost::generateMessage()
{
    // Produce source and random destination addresses.
    ...

    // Create message object and set source and destination field.
    SimpleMessage *msg = new SimpleMessage(msgname);
    msg->setSource(src);
    msg->setDestination(dest);
    return msg;
}

void SimpleHost::forwardMessage(SimpleMessage *msg)
{
```

```
    // Increase hop count
57  ...

    // Randomly select a next hop k
60  ...
    ev << "Forwarding message " << msg << " on port out[" << k << "]\n";
    send(msg, "out", k);
63
}
```

Note that the handleMessage() function has been defined with a pointer to an object of generic class cMessage as argument. Here we use objects of class SimpleMessage instead. So as to have access to specific fields of the SimpleMessage class, it is required to cast the pointer to this more precise object type.

The check_and_cast template method allows to do this safely, i.e. when trying to cast to an improper destination type, the simulation will crash and stop with an error message reporting the wrong casting attempt.

4.5 Simulation configuration, build and run

Change the omnetpp.ini file so that all host[*] nodes have a limit value of 10.

You can now compile and run the simulation. You should see messages generated by host[0] first, then by the final destination of the previous message.

When a host reaches its transmission limit, it stops forwarding and thus the simulation should terminate as no further event remains.

5 Making it wireless

5.1 Objectives

So far, we only modeled a wired network. In this step, we will move to a wireless network model by using the Mobility Framework for OMNeT++ .

Therefore we will use the standard host model bundled with Mobility Framework to set up a simple simulation.

5.2 Network model of Mobility Framework

When using the Mobility Framework , the network should be composed of:

- a ChannelControl module that will manage the network connectivity based on a physical propagation model and distance between hosts
- one or several Host modules, a compound module type that will model a wireless node from application to physical layer.

The ChannelControl module is requiring two parameters, playgroundSizeX and playgroundSizeY, that will define the dimensions of the simulation area.

Therefore, the Network.ned file should be changed as follows:

Listing 13: Network.ned

```
import
    "Host",
    "ChannelControl";

module Sim
    parameters:
        playgroundSizeX : numeric const,
        playgroundSizeY : numeric const,
        numHosts: numeric const;

    submodules:
        channelcontrol: ChannelControl;
            parameters:
                playgroundSizeX = playgroundSizeX,
                playgroundSizeY = playgroundSizeY;
            display: "i=block/control;p=50,25";

        host: Host[numHosts]
            display: "i=device/wifilaptop";

    connections nocheck:
    display: "p=0,0;b=$playgroundSizeX,$playgroundSizeY,rect;o=white";
endmodule

network sim : Sim
    parameters:
        playgroundSizeX = input(40,"playgroundSizeX"),
        playgroundSizeY = input(40,"playgroundSizeY"),
        numHosts =input(5,"Number of hosts:");
endnetwork
```

Note that:

- `Host` is no longer a simple module, but a compound module. Its definition is now in the `Host.ned` file, which is imported at beginning of the file;
- The `ChannelControl` module is part of the Mobility Framework modules and should also be imported.
- As network is wireless and nodes are mobile, it is impossible to define static connections between nodes. So the `connections nocheck` keyword allow to setup the network without prior knowledge of the connections (they will be dynamically managed by the `ChannelControl` module).

5.3 Host model of Mobility Framework

The Mobility Framework models each host from application to physical layer using the following modules:

- `appl`, the application layer generating traffic;
- `net`, the network layer responsible for routing;

- `nic`, which models medium access control (MAC) and impact of physical layer on data delivery;
- `blackboad`, that provides inter-module communication (cross-layering)
- `mobility`, that manages the node mobility;
- `arp`, that provides simple bidirectional MAC to network address conversion.

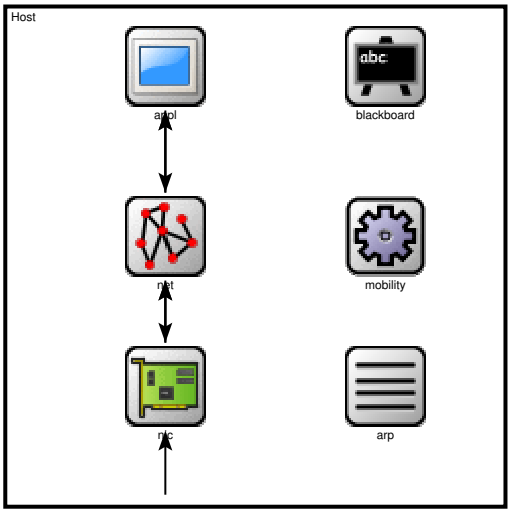

Figure 2: Mobility Framework host model

In this example, we will use built-in models for each of those layers. You are encouraged to go and check the source code of the following modules so as to understand better their behavior and implementation.

The `Network.ned` file to use is the following, and is a typical example of Mobility Framework host configuration:

Listing 14: Host.ned

```
import
    "BurstApplLayer",
    "SimpleNetwLayer",
    "Nic80211",
    "SimpleArp",
    "ConstSpeedMobility",
    "Blackboard";

module Host

    gates:
    in: radioIn; // gate for sendDirect

    submodules:
        blackboard: Blackboard;
            display: "i=block/blackboard;p=200,40";
```

```
18          mobility: ConstSpeedMobility;
                display: "i=block/cogwheel;p=200,125";

21          appl: BurstApplLayer;
                display: "i=block/app2;p=90,40";

24          arp: SimpleArp;
                display: "i=block/table;p=200,200";

27          net: SimpleNetwLayer;
                display: "i=block/network2;p=90,125";

30          nic: Nic80211;
                display: "i=block/ifcard;p=90,200";

33      connections:
            nic.uppergateOut --> net.lowergateIn;
            nic.uppergateIn <-- net.lowergateOut;
36          nic.upperControlOut --> net.lowerControlIn;

            net.uppergateOut --> appl.lowergateIn;
39          net.uppergateIn <-- appl.lowergateOut;
            net.upperControlOut --> appl.lowerControlIn;

42          radioIn --> nic.radioIn;

        display: "p=10,10;b=250,250,rect;o=white";
45  endmodule
```

5.4 Simulation configuration, build and run

So as to configure the simulation correctly, delete all parameters present in the `omnetpp.ini` file and simply add `include mobility-fw.ini`. This will include all configuration parameters defined in the `mobility-fw.ini` file you have copied from the `Development` folder.

Now build the simulation using `make clean`, `opp_mf_makemake` (to include all required dependencies of the Mobility Framework) and `make`. Then launch the newly created executable.

You can observe the simulation running. You may observe that despite the fact that `ConstSpeedMobility` is used, no node is moving. Try to find the explanation for this fact.

Observe the simulation and understand:

- What messages are generated at the application layer?
- What does this application layer do?
- Are the messages unicast, broadcast?
- What messages are generated at lower layers?
- What mechanism is used for channel access?

You can also use the source code of the different modules to answer those questions.

6 Collecting and analyzing statistics

6.1 Objectives

In this part, we will learn how to derivate a module from another one, without the need of creating a NED file to describe it. This is allowed when the child module has same parameters and same gates as its parent (i.e. only internal processing differ).

We will do so in order to add statistics collection features to the application and network layer.

6.2 Child modules definition

In the `Host.ned` file, define two new parameters of type *string*, named `applLayer` and `netwLayer`;

Then, in the submodules description of the `Host` modules, modify the definition of the application and network layers as follows:

Listing 15: Host.ned

```
    parameters:
        applLayer: string,
        netwLayer: string;
...
        appl: applLayer like BurstApplLayer;
            display: "i=block/app2;p=90,40";
...
        net: netwLayer like Flood;
            display: "i=block/network2;p=90,125";
```

Finally, define the value for those parameters in the `Network.ned` file:

Listing 16: Network.ned

```
        host: Host[numHosts]
            parameters:
                applLayer = "StatsBurstApplLayer",
                netwLayer = "StatsFlood";
```

6.3 Child modules implementation

Create the required files to implement two child modules (`StatsBurstApplLayer` and `StatsFlood` respectively).

For a child module implementation to be valid, the following things should appear in the child class definition:

- The child class should publicly extend the parent class;

- The `Module_Class_Members(ChildModule, ParentModule, 0);` function should appear in the class public methods definition;
- The `Define_Module_Like(ChildModule, ParentModule);` function call should appear at the beginning of the `.cc` file.

You should perform these for `StatsBurstApplLayer` and `StatsFlood`.

6.4 Statistics collection

6.4.1 Messages sent/received

So as to collect the number of messages sent and received at the application level, we will use the `StatsBurstApplLayer` child module.

In this module, insert two attributes, `numSent` and `numReceived`, and monitor their evolution during the simulation (using the `WATCH()` command).

Then re-implement the `handleSelfMsg()` and `handleLowerMsg()` methods to increment send and received values when needed. In each case, you will have to do a different processing depending on the message kind. The following code snippet might help you :

Listing 17: StatsBurstApplLayer.cc

```
void StatsBurstApplLayer::handleXXXXMsg(cMessage *msg)
{
    // Perform statistics collection
    ... // numSent++ ? numReceived++ ?
    switch( msg->kind() )
    {
        case ...
            .. // numSent++ ? numReceived++ ?
            break;
        default:
            break;
    }
    // Call parent class method for actual message handling
    BurstApplLayer::handleXXXXMsg(msg);
}
```

Finally, you need to create a method `finish()` in `StatsBurstApplLayer` in order to record the values to the scalar file `omnetpp.sca`.

You can find an example of statistics collection in the `finish()` method in the Tic-Toc tutorial.

6.4.2 Message hop count

In `StatsFlood` create two vector objects so as to store the mean hop count of received packets :

- `cLongHistogram hopCountStats;`
- `cOutVector hopCountVector;`

As done for `StatsBurstApplLayer`, re-implement the `handleLowerMsg()` method to collect statistics before calling the `handleLowerMsg()` of the parent class.

Note that you will have to convert the received message to the type `NetwPkt` and make a decision based on its fields before recording the statistic.

Finally, you need to create a method `finish()` in `StatsFlood` in order to record the values to the scalar file `omnetpp.sca` and vector file `omnetpp.vec`.

You can find an example of **statistics** collection in the `finish()` method in the Tic-Toc tutorial.

6.5 Statistics visualization

Compile and run the simulation. Once it is done, you can use the `scalars` and `plove` commands to analyze the content of the scalar and vector file respectively.

Use those tools to graphically:

- Display an histogram of the sent and received messages for each host;
- Display an histogram of the hop count mean and standard deviation for each host;
- Display the hop count evolution for hosts 5 and 10.

... That's all folks !

Next time:

Now that we have a wireless network model running, we will focus on designing and implementing a more complicated protocol for ad-hoc services.

Based on the protocol description, you will have to model it and implement it into the simulator.

This work will be used in all future labs, with statistics collection, to assess the application behavior in different contexts and find ways for improvement.