MiXiM Tutorial

Tobiáš Smolka

October 11, 2011

Abstract

During the tutorial you should get familiar with basic concepts of OMNet++ and MiXiM frameworks as well as run your first MiXiM based WSN simulation.

1 Getting started

This section briefly introduces technologies and runtime environment, which will be used during the tutorial.

1.1 OMNet++

OMNet++ is a object-oriented modular discrete event network simulation framework, which provides complete environment for building network simulations [2]. The simulation models are implemented by third-party simulation frameworks such as MiXiM, Castalia, INET, ...

OMNeT++ is not a simulator, but provides infrastructure and tools for writing simulations. The component architecture allows to build simulations from reusable components - modules. Modules have parameters to customize their behaviour and communicate with each other via messages. The functionality is implemented in so called simple modules, that are programmed in C++ and make use of OMNet++ API. The simulations can be run under either graphical or command-line user interface.

Most of OMNet++ features are documented in [2].

1.1.1 NED language

OMNet++ uses NED language for defining structure of simulation model (model topology). NED supports defining module parameters (also the default values, units), interface (parameters and gates), metadata annotations and more. NED is also used to create compound modules and interconnect the modules via channels.

Sample NED file can be seen in Figure 1. The file defines a network named `SampleApp` with some parameters (`playgroundSizeX`, ...), one metadata anotation (`@display`), submodules (`connectionManager`, `world`) and an array of submodules (`node`).

1.1.2 Configuring simulations

When the simulation is started in OMNet++, it first reads the structure of simulation model from NED files and then it reads a configuration file (usually `omnetpp.ini`). The file contains

Figure 1: Sample NED file

```
package sampleapp;

import org.mixim.base.connectionManager.ConnectionManager;
import org.mixim.base.modules.BaseWorldUtility;

network SampleApp
{
    parameters:
        double playgroundSizeX @unit(m); // x size of the area the nodes are in (in meters)
        double playgroundSizeY @unit(m); // y size of the area the nodes are in (in meters)
        double playgroundSizeZ @unit(m); // z size of the area the nodes are in (in meters)
        double numNodes; // total number of hosts in the network

        @display("bgb=$playgroundSizeX,$playgroundSizeY,white;bgp=0,0");
    submodules:
        connectionManager: ConnectionManager {
            parameters:
                @display("p=150,0;b=42,42,rect,green,,;i=abstract/multicast");
        }
        world: BaseWorldUtility {
            parameters:
                playgroundSizeX = playgroundSizeX;
                playgroundSizeY = playgroundSizeY;
                playgroundSizeZ = playgroundSizeZ;
                @display("p=30,0;i=misc/globe");
        }
        node[numNodes]: Host802154 {}
    connections allowunconnected:

}
```

settings for the model and also for the simulation core. This includes setting specific values for model parameters, how many runs should be executed, RNG settings, etc.

Sample configuration file can be seen in Figure 2.

1.2 MiXiM

MiXiM (mixed simulator) is an OMNeT++ modeling framework created for mobile and fixed wireless networks (wireless sensor networks, body area networks, ad-hoc networks, etc.). It offers models of radio wave propagation, interference estimation, radio transceiver power consumption and wireless MAC protocols [3].

1.3 Runtime environment

Virtual image has pre-installed Ubuntu 11.04 with OMNet++ and MiXiM installed from source. Login and password is *d*emo.

For this tutorial, we will use OMNet++ v4.1 and MiXiM v2.1. The installation on supported platforms (Windows, Linux, Mac OS X) is covered in OMNeT++ Installation Guide [1] and MiXiM Wiki [4].

2 Simulation `SampleApp`

This part of a tutorial is focused on configuring, executing and analysing results of a simple MiXiM simulation and implementing custom simple module.

The simulation is build from following MiXiM modules:

Figure 2: Sample configuration file (`omnetpp.ini`)

```
[Config experiment_dist_vs_rss]

cmdenv-module-messages = false
cmdenv-express-mode = true
sim-time-limit = 2s

**.playgroundSizeX = 100m
**.playgroundSizeY = 100m
**.playgroundSizeZ = 0m
**.numNodes = 2

**.node[0].isBaseStation = true
**.node[0].mobility.x = 0
**.node[0].mobility.y = 0
**.node[0].nic.phy.last*.scalar-recording = true

**.node[1].mobility.x = 0
**.node[1].mobility.y = ${nodeY=0..60 step 0.5}
**.node[1].net.sinkAddr = 1
**.node[1].net.nextHopAddr = 1

**.node[*].net.readFromFile = false
**.node[*].mobility.readFromFile = false
**.node[*].nic.phy.sensitivity = -100dBm
**.node[*].nic.phy.coreDebug = true
**.node[*].nic.phy.useThermalNoise = false

**.node[*].appl.packetSendInterval = 1s
**.node[*].appl.numPackets = 1

**.scalar-recording = false
**.vector-recording = false
```

- **SensorApplLayer** - periodically sends data packet to the sink.
- **WiseRoute** - implements simple routing scheme.
- **Nic802154_TI_CC2420** - implements IEEE 802.15.4 network interface.
- **BaseMobility** - implements static mobility scheme.

2.1 Creating the simulation

1. After starting the VM, open the terminal and execute command `omnetpp`. This will start the OMNet++ IDE, an Eclipse based IDE with integrated support for OMNet++ projects.
2. Create new OMNet++ project (`File` → `New` → `OMNet++ Project`).
   (a) Enter project name `SampleApp`, click `Next`.
   (b) Select `MiXiM` → `Basic MiXiM network`, click `Next`.
   (c) Select `Sensor Application Layer`, `WiseRoute`, `CSMA 802.15.4`, `Static (no mobility)` and `2-dimensional`, click `Finish`.
3. Build and run the simulation
   (a) Right click on project `SampleApp`, select `Build project`.
   (b) Right click on project `SampleApp`, select `Run As` → `OMNet++ Simulation`.
   (c) Explore Tkenv (OMNet++ graphical user interface) a little bit and try out how `Run`, `Stop`, `Speed` and `Zoom` buttons work.
   (d) Close Tkenv and finish the simulation.

2.2 Modifying the simulation

The `SampleApp` simulation created in previous step utilizes mentioned modules in a way, that all nodes are broadcasting the packets and flood the network. However, in order to prepare more interesting scenario with hop-by-hop routing to the sink, there is no need to reprogram the model. All the changes can be made in `omnetpp.ini` configuration file.

Open file `omnetpp.ini` and perform following modifications ({m} modify line, {+} add new line, {-} remove old line, {=} no change):

1. Configure detailed logging of all events (will be used for follow-up analysis of the simulation results).

```
{=} **.**.coreDebug = false
{+} record-eventlog = true
```

2. Change playground size and number of nodes.

```
{m} **.playgroundSizeX = 500m
{m} **.playgroundSizeY = 500m
{m} **.playgroundSizeZ = 500m
{m} **.numNodes = 7
```

3. Disable broadcast, set `node 0` as a sink and configure `node 1` to generate high amount of packets.

```
{=} **.node[*].applType = "SensorApplLayer"
{+} **.node[1].appl.nbPackets = 9999
{=} **.appl.trafficType = "periodic"
{=} **.appl.trafficParam = 1 #in seconds
{m} **.appl.broadcastPackets = false
{+} **.appl.destAddr = 0
{m} **.appl.nbPackets = 0
{+} **.appl.trace = true
{+} **.appl.debug = true
```

4. Change mobility parameters and place all nodes except sink randomly.

```
{m} **.node[0].mobility.x = 50
{m} **.node[0].mobility.y = 50
{-} **.node[0].mobility.z = 250

{-} ... all lines with mobility params for other nodes ...

{+} **.node[*].mobility.x = -1
{+} **.node[*].mobility.y = -1
{+} **.node[*].mobility.z = -1
```

After the modification, the simulation should contain two phases:

1. **Route discovery phase** - The sink (`node 0`) broadcasts `route-flood` message, which will be propagated to every node in the network. During this phase, each node computes its own routing table for routing to the sink.
2. **Data sending phase** - The `node 1` will periodically (each second) send data packet to the sink. Since there is a initial waiting period and the route discovery phase proceeds data sending phase, the packet should be successfully routed to the sink.

Test the modified simulation, make sure the animation speed is low and observe the two phases. Also try out following actions.

- Double click on the `node 0` to see the inner composition of the module. Wait for a data message and see how it is propagated all the way up to the application layer.
- Stop the simulation while there is some AirFrame being send between two modules, find it (in NIC module of the receiver) and inspect it to see the encapsulation and specific parameters (i.e. different `destAddr`, `srcAddr` on application and network level).
- Find the application layer of the `node 0` and perform `Inspect module output` action. Increase the speed of the simulation and observe logging messages from application layer about receiving the packets.
- Go in the application layer of the `node 0`, find `cOutVector rawLatencies`, open it and observe how the latency of the received packets is changing in real-time.

Stop the simulation after at least 50 simulated seconds and make sure the `finish()` method is called so the data for follow-up analysis is written down in the result files.

2.3 Analysing the results

OMNet++ provides support for recording outputs of the simulation via output `vectors`, `scalars` and `histograms`. Previous simulation collected number of statistics. A vector was used to record latencies of received data packets and a scalar was used to record total amount of received data packets. Additionally, the simulation also recorded log of all events, that were created during the simulation.

Before performing next steps make sure the simulation is stopped and there are output files generated in `results` folder of the project (*.sca, *.vec, *.elog).

Inspect result vectors and scalars:

1. Double click on either *.sca or *.vec output file and confirm creation of new analysis file (in case it was not created before).
2. Browse the data, try out filtering by statistic name (e.g. `rawLatencies` in vectors, `nbData*` in scalars).
3. Select filtered vectors or scalars and perform `Plot` action.
4. Use `scavetool` for exporting scalars to CSV file.
   (a) Open the terminal and enter following commands:
   ```
   cd Projects/SampleApp/results/
   scavetool scalar -p "name(nbData*)" *.sca -O stats.csv
   cat stats.csv
   ```

Inspect event log:

1. Double click on *.elog output file and open event log window.
2. Apply filter by module NED type `org.mixim...SensorApplLayer` to see only application level events (sending and receiving data packets).
3. Closely inspect event #358 (arrival of first data message into sink) by applying "Causes/Consequences" filter.
   (a) Discard filtering by module NED type and apply "Causes/Consequences" filter for event 358.
   (b) Notice the chain of events starting with `route-flood` via nodes `0`, `6`, `4`, `1` and ending with data packet transmission via nodes `1`, `4`, `6` and `0`.

Figure 3: Listing of file `EvilNetw.ned`

```
package sampleapp;

import org.mixim.modules.netw.WiseRoute;

simple EvilNetw extends WiseRoute
{
    parameters:
        double pDataPacketDropping = default(0.5);  // probablity of dropping a packet
        @display("i=block/cogwheel");               // meta data for displaying an icon
        @class(EvilNetw);                           // reference to implementing class
}
```

Figure 4: Listing of file `EvilNetw.h`

```
#ifndef __SAMPLEAPP_EVILNETW_H_
#define __SAMPLEAPP_EVILNETW_H_

#include <WiseRoute.h>

class EvilNetw: public WiseRoute {
  protected:
    long nbDataPacketsDropped;                  // number of dropped packets
    virtual void initialize(int);               // initialize module
    virtual void finish();                      // finalize module
    virtual void handleLowerMsg(cMessage*); // handle message
};
#endif
```

2.4 Implementing a simple module

The functionality of the new module is quite simple – it adds a support for selective forwarding attacks into original `WiseRoute` network layer.

It implements method `handleLowerMsg()` for intercepting all messages from lower layer (MAC layer) (see Figure 5). The module selectively drops all data packets with a probability configurable in module parameter `pDataPacketDropping`. In case the message is dropped, the module also appends a message into the log, display notification "bubble" (in Tkenv) and increment `nbDataPacketsDropped`. During the finalization, `nbDataPacketsDropped` is recorded to output file as a scalar.

1. Create new simple module by `File` → `New` → `Simple Module`.
   (a) Enter `EvilNetw.ned` as a name of NED file, click `Finish`.
2. Modify content of generated files `EvilNetw.ned` (see Figure 3), `EvilNetw.h` (see Figure 4) and `EvilNetw.cc` (see Figure 5). All files can be downloaded from [5] (copying from PDF will not work).
3. Open `omnetpp.ini` and perform following changes ({m} modify line, {+} add new line). The modifications will force `node 6` to use `EvilNetw` instead of `WiseRoute`.

```
[+] **.node[6].netwType = "EvilNetw"
[=] **.node[*].netwType = "WiseRoute"
```

3 Simulation `basicIDS`

Project `basicIDS` models a prototype of simple IDS. The IDS runs on each node in the network and is capable of detecting selective forwarding and hello flooding attackers. For this purpose, almost all modules in the network had to be extended with specific functionality.

Figure 5: Listing of file `EvilNetw.cc`

```
#include "EvilNetw.h"
Define_Module(EvilNetw);

// Initializes EvilNetw module
void EvilNetw::initialize(int stage){
  WiseRoute::initialize(stage);
  if(stage == 1) {
    nbDataPacketsDropped = 0;
  }
}
// Finalize module and record statistics
void EvilNetw::finish(){
  WiseRoute::finish();
  if (stats) {
    recordScalar("nbDataPacketsDropped", nbDataPacketsDropped);
  }
}
// Handle message from lower (MAC) layer
void EvilNetw::handleLowerMsg(cMessage* msg) {
  // Consider only DATA packets and drop only with certain probability
  if (msg->getKind()==DATA && par("pDataPacketDropping").doubleValue()>=uniform(0,1)){
    getNode()->bubble("Dropping a packet!"); // notification
    EV << "Dropping a packet!" << endl; // logging
    nbDataPacketsDropped++; // statistics
    delete msg; msg = NULL; return;
  }
  WiseRoute::handleLowerMsg(msg); // fallback
}
```

The tutorial does not provide detailed analysis of each aspect of the simulation model, instead it demonstrates 2 simple experiments: `test` and `experiment_dist_vs_rss`.

3.1 Experiment `test`

Experiment `test` is configured in the `test` section of `omnetpp.ini`. The network contains one base station and 4 nodes. The nodes are periodically (each second) sending packets to the base station, the total number of send packets for each node is `300`.

First, run the scenario, in which all nodes are simply sending the packets to the base station.

1. Run the experiment with initial settings and observe the behaviour of the network.
   - Note: the radio range of the nodes is smaller than in the previous simulation, use `Zoom in` and `Decrease icon size` actions (Ctrl+M, Ctrl+O).
2. Change routing in the network and observe performance of IDS detection of selective forwarding attacks.
   (a) Set routing of `node 1` via `node 2` and check IDS performance.
   ```
   [+] **.node[1].net.nextHopAddr = 2
   [=] **.node[1..4].net.nextHopAddr = 0
   ```
   (b) Open result file `test-0_ids_forwarders_stats.txt`, which contains IDS statistics in the following format: A;B;PR;PF;DIST;DROPPED, where
   A is a monitoring node, B is a monitored node,
   PR is number of received packets by the node B,
   PF is number of forwarded packets by the node B,
   DIST is a distance between nodes A and B,
   DROPPED is packet dropped ratio.
3. Add an attacker to the network and again check IDS performance.
   (a) Configure `node 2` as a selective forwarder with dropping probability 0.5.

```
[=] **.node[1].net.nextHopAddr = 2
[+] **.node[2].appl.attacker = "selectiveForwarder"
[+] **.node[2].appl.pPacketDropping = 0.5
```

(b) Open again result file `test-0_ids_forwarders_stats.txt` and check performance of the detection on nodes `0` and `1`.

3.2 Experiment `experiment_dist_vs_rss`

The experiment is defined in section `experiment_dist_vs_rss` of the configuration file `omnetpp.ini`. Its goal is to determine how is the received signal strength affected by the distance between sender and receiver.

In order to simulate such a scenario, there are 2 nodes in the network. One base station and a node, that sends only 1 packet. Instead of modelling movement of the node (which is also possible), the experiment includes 121 runs with different position of the node. All the runs are defined by following statement: `**.node[1].mobility.y = $nodeY=0..60 step 0.5`

In this setting, the parameter `y` of module `mobility` of `node 1` will have values 0, 0.5, 1, ... 60 subsequently in experiment runs 0, 1, 2, ... 121.

Although all the runs could also be executed via IDE, it is recommended to execute them from command line due to the performance reasons.

1. Open the terminal and enter following commands:

   ```
   cd Projects/basicIDS/
   ./run_experiment_dist_vs_rss.sh
   ```

2. Open the IDE and open the analysis file `experiment_dist_vs_rss.anf`.

3. Go to `Datasets` tab of the file and display chart `Distance vs. RSS`.

References

[1] *OMNeT++ Installation Guide, Version 4.1*, `http://omnetpp.org/doc/omnetpp41/InstallGuide.pdf`

[2] *OMNeT++ User Manual, Version 4.1*, `http://www.omnetpp.org/doc/omnetpp41/Manual.pdf`

[3] *MiXiM project*, `http://mixim.sourceforge.net/`

[4] *MiXiM Wiki*, `http://sourceforge.net/apps/trac/mixim/wiki`

[5] *MiXiM Tutorial Wiki*, `https://minotaur.fi.muni.cz:8443/~xsvenda/docuwiki/doku.php?id=public:wsn:mixim_tutorial`