A Mobility Framework for OMNeT++ User Manual

Version 1.0a4

Marc Löbbers
Daniel Willkomm
`loebbers|willkomm@tkn.tu-berlin.de`

January 12, 2007

WARNING: This is the manual of the **OLD** Mobility Framework, version 1. There are serveral parts wich are not up tp date and have changed in version 2. However, it might still be helpfull to understand the general concepts of the Mobility Framework.

Contents

1 Introduction **3**
1.1 What is the Mobility Framework for? 3
1.2 History 3
1.3 Authors 4

2 Overview **4**
2.1 Concept 4
2.2 Using the Mobility Framework 5
2.3 Directory structure 7

3 Getting the Mobility Framework started **8**
3.1 Installation 8
3.2 Creating an own network 8
3.3 Creating a new module 9

4 Building Own Simulations **10**
4.1 The Basic Module concept 10
4.2 The Message Concept 13
4.3 The Nic Concept 16
4.3.1 snrEval 17

5 Physical Layer Modules **17**
5.1 SnrEval 18
5.2 Decider 19
5.3 P2PPhyLayer 19

6 Using the Blackboard **19**
6.1 Methods for the subscriber 20
6.2 Getting informed 21
6.3 Methods for the publisher 22

7 Mobility Modules **24**
7.1 The Mobility Architecture 24
7.2 ChannelControl 25
7.3 Implementing Mobility Models 26

1 Introduction

1.1 What is the Mobility Framework for?

This framework is intended to support wireless and mobile simulations within OMNeT++. The core framework implements the support for node mobility, dynamic connection management and a wireless channel model. Additionally the core framework provides *basic modules* that can be derived in order to implement own modules. With this concept a programmer can easily develop own protocol implementations for the Mobility Framework (MF) without having to deal with the necessary interface and interoperability stuff.
The framework can be used for simulating:

- fixed wireless networks
- mobile wireless networks
- distributed (ad-hoc) and centralized networks
- sensor networks
- multichannel wireless networks
- many other simulations that need mobility support and / or a wireless interface

We are currently developing a library of standard protocols for the MF (802.11, AODV, ...). Our goal is to have a rich library of such protocols to enable easy plug-and-play simulations of various kinds of widely used protocols.

1.2 History

This mobility extension for OMNeT++ has been written by several people at the Telecommunication Networks Group at the Technische Universitaet Berlin.

The first version (FraSiMO) was written within a student project in June 2001 by Heiko Scheunemann and Daniel M. Kirsch. However, none of their code survived the complete review by Steffen Sroka in 2002/2003, who rewrote the framework for better integration with OMNeT++ and higher speed (FraSiMO II). He had help from Christian Frank, who reviewed the mobility and position handling and Witold Drytkiewicz who speeded up the connection handling of the ChannelControl module.

The current version (mobility-fw) was started in October 2003 by Marc Loebbers. He added the dynamic gate handling and started structuring and documenting the code. He had a lot of help from Daniel Willkomm. During this process the

ChannelControl and *Blackboard* were completely rewritten by Andreas Koepke. The current version of the *Blackboard* is a result of an attempt to define common interfaces for the Mobility Framework and the *IP-Suite* and is maintained by Andras Varga.

This first attempt of a manual was written by Marc Loebbers and Daniel Willkomm. It is based on the paper ”A Mobility Framework for OMNeT++” presented at the third International OMNeT++ Workshop at Budapest University of Technology and Economics in January 2003.

1.3 Authors

The general idea and overall structure is mainly due to
Holger Karl (`karl@tkn.tu-berlin.de`).

FraSiMO:
Heiko Scheunemann TU Berlin
Daniel M. Kirsch TU Berlin

FraSiMO II:
Steffen Sroka TU Berlin
Christian Frank TU Berlin
Witold Drytkiewicz TU Berlin

Mobility Framework (mobility-fw):
Marc Loebbers TU Berlin (`loebbers@tkn.tu-berlin.de`)
Daniel Willkomm TU Berlin (`willkomm@tkn.tu-berlin.de`)
Andreas Koepke TU Berlin (`koepke@tkn.tu-berlin.de`)

2 Overview

2.1 Concept

The two core components of the Mobility Framework (MF) are an architecture for mobility support and dynamic connection management and a model of a mobile host in OMNeT++. Figure 1 shows a network setup with 10 nodes.

The *ChannelControl* module controls and maintains all **potential** connections between the hosts. An OMNeT++ connection link in the MF does **not** automatically indicate that the corresponding hosts are able to exchange data and communicate with each other. The *ChannelControl* module only connects all hosts that *possibly* interfere with each other. A communication link is probably easiest defined by its complement: *All hosts that are* ***not*** *connected definitely do not interfere with each other*. Following this concept a host will receive every data packet that its transceiver is potentially able to sense. The physical layer then has to decide dependent on the received signal strength whether the data packet will be processed or whether it will be treated as noise. For further details on the connection management please refer to Section 7.2.

The internal structure of a mobile host is shown in Figure 2. Apart from the standard ISO/OSI layers there is also a *Mobility* module and a module called *Blackboard*. The *Mobility* module provides a geographical position of the host and handles its movement. Section 7.1 contains a detailed description of the mobility architecture. The *Blackboard* module is used for cross layer communication. It provides information relevant to more than one layer like the actual energy status of the host, the display appearance or the status of the radio. All other modules implement the corresponding ISO/OSI layer functionality. Details on the implementation of these modules can be found in Section 4.

While the core MF does not provide any protocol implementations we also plan to provide a library of standard modules for the lower layers of the ISO/OSI protocol stack. Thus the MF will eventually enable simulation of various kinds of wireless mobile networks “out of the box”.

2.2 Using the Mobility Framework

There are several ways to use the Mobility Framework. Which one is best suited depends on the purpose you want to use the MF for. Once you successfully compiled the MF you can just use the created library within your own simulation or you should take a look at the example networks in the `networks/` and `core/basicNetworks` directories if you want to create a new simulation based on the Mobility Framework. Please note that you need to add the `lib` directory to your `LD_LIBRARY_PATH` or copy the library to a location where the linker can find it in order to work with the MF.

We provide example networks for many different scenarios so it is very likely that you will find a setup similar to your needs. The networks and the protocol implementations used are very well documented in the API documentation (`doc/api/index.html`) so you should take a look there as well.

The `template/` folder contains templates for the derivation or creation of your own modules. You can copy the corresponding files you need, adapt them to your needs and of course fill them with your code. Copy the `Makefile.gen` file as well and change the `MOBFW` variable within `Makefile.gen` to point to your `mobility-fw` folder and you can easily create a Makefile for your files (just type `opp_makemake -f Makefile.gen`).

Please note that the mobilityfw library does not contain any modules from the `development/` folder. Those implementations are to be considered experimental so we did not include them into the library. So if you want to enhance one of the protocol implementations or use them as a base for you own implementation you would have to copy the corresponding files as well. An alternative would be to do your work within the `development/` directory structure. You will have to run `make makefiles` in the corresponding directories to "register" your newly created classes with the Makefiles.

We explicitly encourage the use and improvement of those protocol implementations under development. The more feedback and bug-fixes we get the sooner those protocols will reach a "stable" status and can be included into the library. Additionally many of them already reached a decent state and only need some more testing so please also report if you use those implementations and do not experience problems. Please refer to the corresponding API documentation and contact the individual authors for questions, feedback and bug reports.

2.3 Directory structure

Here is a list of all different directories including a brief description of the content to help you find certain files.

3 Getting the Mobility Framework started

This section will explain the basic use of the MF. We assume that you are familiar with programming in OMNeT++. If not you should read the OMNeT++ manual (*http://www.omnetpp.org/*). After explaining how to install the MF we will give a guideline how to create an own simulation network and how to derive your own modules. For a description of the functionality of the different modules please read Section 4 and the API documentation.

3.1 Installation

You need a running OMNeT++ version 3.0a8 or higher (recommended is 3.0a9 or higher) to use the MF with all of its functionality. After downloading the most recent version of the MF from the download area of *http://mobility-fw.sourceforge.net* copy the file to the desired directory. `cd` to this directory and

- `tar xzf mobility-fw-`*version*`.tgz`
- Add `your/path/to/mobility-fw-`*version*`/lib`
  to your `LD_LIBRARY_PATH` variable

```
mobility-fw/         root directory
  bitmaps/           icons for network graphics
  core/              the core of the framework
    basicMessages/   the basic message classes
    basicModules/    all basic module classes
    basicNetworks/   2 basic sample networks
    blackboard/      Blackboard stuff
    channelcontrol/  ChannelControl modules
    utils/           utilities
  development/       experimental protocol implementations
    applLayer/       application layer modules
    messages/        all .msg files
    mobility/        Mobility modules
    netwLayer/       network layer modules
    nic/             network interface modules
      macLayer/      MAC layer modules
      phyLayer/      physical layer modules
        decider/     decider modules
        snrEval/     snr evaluation modules
    utils/           utilities
  doc/               manual, API, neddoc...
  include/           header and .ned includes
  lib/               dir of the libmobilityfw.so
  networks/          example networks
  protocols/         tested protocol implementations
  template/          templates to create own modules
  testSuite/         regression test suite
```

- `cd mobility-fw-`*version/*
- `make install`
- `make`

where *version* is the version number of the file you downloaded. You should run the example simulations in the `core/basicNetworks` directory to see if everything was built properly.

The file `doc/hp/index.html` will link you to the documentation (API and Neddoc reference and this manual) for the MF. You can also find links to coding and documentation guidelines for your implementations on this page.

3.2 Creating an own network

To be able to easily create a new simulation network we provide template files. They can be found in the `template/` directory. Basically all you have to do is to copy the desired template files into your network directory and adapt them. The most important advises and rules are presented as comments in these files.

To create a new network you should create a new directory with the name of *yourNetwork*. Copy the files *YourNetwork.ned*, *YourHost.ned* and *Makefile.gen* (and *YourNic.ned* if needed) to the network directory. Give them names of your choice and also adapt the module names inside these files correspondingly. If you want to create own modules also copy the corresponding template files to the *yourNetwork* directory, give them proper names and fill them with functionality as described in Section 3.3.

You also have to set the `MOBFW` variable in the *Makefile.gen* to the path of your *mobility-fw* directory so that the Makefile can find the MF library header and ned files.

In the file *YourHost.ned* you have to declare what NIC and what modules you want to use as protocol layers. If you want to create an own *Nic Module* you also have to copy and modify the *YourNic.ned* file.

As soon as everything is ready you need to run `make -f Makefile.gen` to create a *Makefile* and then run `make`. If you did not make any mistakes ;-) an executable file *yourNetwork* is created. By running it the simulation is started.

3.3 Creating a new module

For every simple module three files have to be created, a *.ned*, a *.h* and a *.cc* file. The best choice is to put all your own modules into the *yourNetwork* directory (see Sec. 3.2). As an example we will assume to create a new MAC module from the *YourMacLayer* template in this section. However the procedure for all other modules is more or less the same.

The template files already contain the main structure and only have to be modified and filled with functionality. The most important hints, rules, advises and suggestions are given in the files as comments.

The *YourMacLayer.ned* file already contains the obligatory gates and the parameters needed by the *BasicMacLayer* module. Just add the additional parameters you need for your module and change the name of the module to the desired one. If you do not derive your module from a *BasicMacLayer* but an already existing protocol implementation (e.g. *Mac802.11.ned*) you have to add the additional parameters needed by this module as well.

The *YourMacLayer.h* file already contains the class definition with the correct derivation of the *BasicMacLayer* class and the *Module_Class_Members()* macro. If you want to derive your module from an already existing protocol implementation you have to replace *BasicMacLayer* (e.g. with *Mac802.11*). The next thing you should do is a "search and replace" of *YourMacLayer* with the desired name for your module. Then this file simply has to be extended with all the extra functions and variables you need. If you want to use the *Blackboard* you have to uncomment the *blackboard** functions (see Sec. 6).

The *YourMacLayer.cc* file contains the basic structure to fill the module with functionality. The first thing you should do is a "search and replace" of *YourMacLayer* with the class name you chose in the header file. The *Define_Module()* macro is also included. If you do not use your own .ned file you have to replace this with the *Define_Module_Like()* macro.

The *handleUpperMsg()* and *handleLowerMsg()* functions contain the *sendUp()* and *sendDown()* functions with example parameters in the obligatory structure. If you want to use the *Blackboard* uncomment the *blackboard** functions and fill them with code. They already contain the structure they should have so please do not change it (see Sec. 6).

4 Building Own Simulations

This section explains the basic concepts behind the Mobility Framework. The class hierarchy is explained and all relevant functions of the *Basic** modules are introduced. For the detailed description of the functionality of the individual modules read Sections 5 - 7 and also refer to the Neddoc and API reference.

4.1 The Basic Module concept

All our *Host* submodule classes have a common base class *BasicModule*. The *BasicModule* itself is derived from *cSimpleModule* (OMNeT++) and *Blackboard-Access* which provides the functionality to subscribe for and publish information on the *Blackboard* module. The usage of the *Blackboard* is explained in Section 6.

BasicModule The *BasicModule* uses the multi-stage initialization of OMNeT++ so all modules in the MF have to be implemented with two-stage initialization. In case you use the *Blackboard* you have to do all your *publish* calls in stage 0 and the *subscribe* calls have to be done in stage 1! This is to explicitly avoid that a subscription to a parameter can happen before that parameter is published.

In the template files the *initialize(int)* functions already contain the call of the *initialize(int)* function of the *Basic Module* this module is derived from. Do not delete this line as the obligatory parameters of the *Basic Modules* have to be initialized, too!

Additionally the *BasicModule* provides the *logName()* function, which returns the ned module name and the OMNeT++ module *index()* of the *Host* module this module belongs to. *logName()* is used in the debug macro for identification of the host the debug output comes from. You can use the *EV ¡¡"your debug output"* macro to print debug messages.

Address Concept We use the OMNeT++ module *id()*s for addressing in the MF. The *nic* module *id()* is used as *mac* address and the *netwLayer* module *id()* as network layer address. The *netwLayer id()* is also used as application layer address. In order to obtain the address of the module we provide a *myApplAddr()*, *myNetwAddr()* and *myMacAddr()* function in the *BasicApplLayer*, *BasicNetwLayer* and *BasicMacLayer* modules respectively.

The *netw2mac()* function of the *ChannelControl* module provides a simple ARP like translation of network addresses to MAC addresses. However by redefining the *getMacAddr()* function of the *BasicNetwLayer* you can implement your own ARP functionality.

Naming Conventions There are some naming conventions for the modules in the ned files, you have to follow. Not following these naming conventions will result in segmentation faults upon execution of your code!!!

All *host* modules **have to** include the characters *host* or *Host* somewhere in its name. Whereas most of the example networks just use *host* as a ned module name, you could also come up with names like *baseStationHost* and *mobileHost* in order to create a centralized base station network.

For more or less all the other ned modules the names **cannot** be changed. Those are: *channelcontrol*, *blackboard*, *net*, *phy*, *snrEval*. However, it is a good practice to also keep the default names for the other modules (namely, *appl*, *nic*, *mobility*).

Basic* **Modules** In order to have clearly defined interfaces that are easy to understand – and extend if necessary – we provide a *Basic** module for each layer which in turn is derived from the global *BasicModule* explained above. The concept of a *Basic** module is to have a base class which takes care of the necessary work that has to be done but is not of specific importance for the real functionality. The general derivation structure of a Mobility Framework module is shown in Figure 3. The *Basic** modules also provide a very basic functionality in terms of "the *BasicNetwLayer* is capable of forwarding messages to and from upper and lower layers but has no routing functionality at all yet". The idea is to have the possibility to easily extend or adapt modules of different layers to the specific requirements of the simulation. To serve this purpose we defined two kinds of functions: *handle*Msg()* functions and *convenience* functions.

handle*Msg() **Functions** The *handle*Msg()* functions contain the actual protocol functionality. They are called each time a corresponding message arrives and contain all necessary processing and forwarding information for messages where required. We provide three different functions to handle the three different kinds of message events possible:

handleSelfMsg The easiest way to implement timers in OMNeT++ are self messages. *handleSelfMsg()* thus is the place to handle all timer related things and to initiate actions upon timeouts.

handleUpperMsg This function is called every time a message has arrived from an upper layer. The message already has the corresponding layer n format (i.e. it is already encapsulated). After processing the message can be forwarded to the lower layers with the *sendDown()* function, if necessary.

handleLowerMsg For messages from lower layers it is the other way around. After being processed they have to be forwarded to upper layers if necessary. This is done by using the *sendUp()* function which also takes care of decapsulation.

convenience **Functions** The *convenience* functions are defined to facilitate common interfaces and to hide inevitable interface management from the user. Here we provide three different functions:

encapsMsg This function is called right after a message has arrived from the upper layers. It is responsible for encapsulation of the layer n+1 message into a layer n message. This implies to provide all necessary header information and if applicable also the conversion of layer n+1 header information into layer n information (in case of the network layer to convert the application or transport layer address into a network address). After this the message is passed to the *handleUpperMsg()* function.

sendUp *sendUp()* is the function to be called if a message should be forwarded to upper layers and is usually called within *handleLowerMsg()*. It decapsulates the message before sending it to the layer n+1 module.

sendDown Sending messages to layer n-1 is done with the *sendDown()* function. Sometimes it may be necessary to also provide or process additional meta information here. In the case of the network layer for example it may be necessary to provide a next hop. The network layer destination address usually contains no information about the next-hop MAC address a message has to be forwarded to on its way to the destination so it has to be translated (ARP does this for IP).

sendDelayedUp In case you want to delay the point of time the message is sent to the upper layer, you can use this function. The time the message should be delayed can be given as a parameter.

sendDelayedDown The same as *sendDelayedUp*, but for sending delayed messages to the lower layer.

These six functions are provided (with slide differences) in the *Basic** module for each layer of the MF. Usually the three *handle*Msg()* and the *initialize(int)* and *finish()* functions are the only functions a programmer has to re-implement to create his/her own module and the functionality of it. The *convenience* functions should be used to serve the above described tasks so that newly implemented modules remain compatible with (almost) any other module implemented for the MF. The *convenience* functions **cannot** be changed in derived modules.

The tasks of the different modules and a more detailed description of the *Basic** modules can be found in the following sections.

4.2 The Message Concept

In order to provide basic functionality like encapsulating and decapsulating messages in the *Basic** modules we need to have fixed message formats for every layer. The provided message types have the most important fields needed for the corresponding layer. These message types with these fields are obligatory and can only be extended but **not** exchanged. Here is a list of all base message classes and their parameters:

- *AirFrame* — Physical Layer Message

pSend	sending power
channelId	channel the message is sent on
id	id of the originator to get the position
duration	time needed to send the message on the channel
SnrControlInfo	Control Information class; used to pass Sn(i)r information to the decider

- MacPkt — Medium Access Control Message

destAddr	destination MAC address
srcAddr	source MAC address
channelId	channel the message is sent on

- *NetwPkt* — Network Layer Message

destAddr	destination network address
srcAddr	source network address
seqNum	sequence number
ttl	time to life
MacControlInfo	Control Information class; used to tell the MAC protocol the address of the next hop

- *ApplPkt* — Application Layer Message

destAddr	destination application address
srcAddr	source application address

Creating your own message If you need additional parameters you can derive your own message class from one of the basic message classes in the OMNeT++ style, in case of the network layer message:

```
cplusplus {{
#include "NetwPkt_m.h"
}};

class NetwPkt;

message YourPkt extends NetwPkt
{
   fields:
        int     extraField1;
        double  extraField2;
};
```

Please note that the name of the included *.h* file has to be extended with *_m* as this is the file OMNeT++ creates out of a *.msg* file.

Using an own message Now you have defined your own message, the next step is to teach the framework how to create it. To do so, you have to overwrite the *createCapsulePkt* function in *YourNetwModule.h*:

```
virtual NetwPkt* createCapsulePkt() {
   return new YourPkt;
};
```

When creating a new *YourPkt* message within your simulation it has to be done like this:

```
YourPkt* pkt = static_cast<YourPkt *>(createCapsulePkt());
```

The framework does the same and is hence able to create *YourPkt* without even knowing that such a thing would spring into existence.

These steps are necessary to be able to do the following unavoidable cast operation. A message is usually given to you as a parameter in the *handleUpperMsg()* and *handleLowerMsg()* function in the basic message format. So, if you want to have access to the extra fields you have to cast the message into your format. For *YourNetwModule* you could use the following code:

```
void YourNetwModule::handleUpperMsg(NetwPkt *packet)
{
  YourPkt *pkt = check_and_cast<YourPkt *>(packet);

  // do something with the message...
}
```

Control Information Classes OMNeT++ (version 3.0a4 or higher) allows you to define *Control Information classes*. They can be used to add meta information to a message that is only relevant for the next processing layer. When that layer receives this message it then can remove and process this information. As you can see in the tables, *AirFrame* and the *NetwPkt* have such a control information class.

The *SnrControlInfo* is used to pass s(i)nr information to the *decider* so that it can calculate bit errors based on the s(i)nr information of that message. The

MacControlInfo contains the next hop MAC address the message should be forwarded to. Currently the MF provides no means to extend the control information attached to a message. If you need to exchange additional meta information you would have to add it as a “normal” parameter to your derived message as only one control information is allowed per message. However please tell us in case you need additional control information as we might extend the *SnrControlInfo* or *MacControlInfo* respectively if appropriate. Currently we only provide the absolutely necessary information. But especially the *MacControlInfo* serves as interface between MAC and network layer that we want to provide as powerful as possible.

4.3 The Nic Concept

A nic is a network interface card that includes physical layer functions like transmitting, receiving, modulation as well as medium access mechanisms. The *nic* module in the MF therefore is divided into a physical layer like part (*snrEval* and *decider*) and a MAC layer (*macLayer*). The *snrEval* module can be used to compute some sn(i)r information for a received message whereas the *decider* module can process this information to decide whether a message got lost, has bit errors or is correctly received. Therefore the decider only handles received messages and not messages that should be sent. The corresponding compound module with its simple modules is shown in Figure 4.

The reason for putting the physical and the MAC components into one compound module is easily explained. For most lower layer protocols the MAC *and* the physical layer have to be coordinated, so for one protocol (e.g. IEEE 802.11) there will be a corresponding *snrEval* module as well as a corresponding *decider* module as well as a corresponding *mac* module. So, if you decide to run a certain routing protocol over a certain PHY/MAC protocol you simply need to choose the corresponding *nic* module when building the host (see Section 3.2).[1]

In the following the physical layer modules are explained in more detail.

4.3.1 snrEval

The structure of the *snrEval* modules is a little different from those of the other modules. The *handleLowerMsg()* function is split into two functions in order to simulate the transmission delay. Detailed examples for using these functions are given in the *YourSnrEval* template files.

[1]Hopefully, there will be several working nics in the library soon, but the *nic* modules defined in the development directory can be used, too!

handleLowerMsgStart This function is called in the moment the reception begins, i.e. in the moment the first bit arrives. Everything that is necessary to be done at the start of a reception can be done here, e.g. create and initialize an SNR-list to store SNR values, put the frame into a receive-buffer etc.

handleLowerMsgEnd This function is called when the transmission of a message is over. Here you can do whatever is necessary before to message is handed on to the decider, e.g. take the message out of the receive-buffer, call the *sendUp* funtion…

5 Physical Layer Modules

The base class for all *PhyLayer* modules is *ChannelAccess* which in turn is derived from *BasicModule*. The only functionality *ChannelAccess* provides is the connectivity to the channel (i.e. to other nodes). The function *sendToChannel()* should be used to pass messages to the channel. It will send the message to every connected gate of the *Host* module.

We provide two versions of a *PhyLayer*. The first version called *P2PPhyLayer* assumes point to point connections between *Hosts* and is described in Section 5.3. It is the simplest *PhyLayer* you can think of and especially useful if detailed propagation and interference models are not needed.

For the second version we decided to divide the physical layer functionality into two submodules. The *PhyLayer* is divided into an *SnrEval* and a *Decider* submodule (see Fig. 4). We wanted to keep the SNR calculation and evaluation separate from the decision about bit errors. This concept makes it very easy to create different *Decider* modules that use the same *SnrEval* module and vice versa. We can define for example a *Decider* module that just compares the calculated SNR with a certain threshold and one that uses forward error correction and both modules can use the same *SnrEval* module.

Typical parameters of an *snrEval* module are *transmitterPower, carrierFrequency* and *pathLossAlpha*. They can be used to compute the attenuation of a signal as well as *snir* values. The *ChannelControl* module also has versions of these parameters (*pMax, carrierFrequency, alpha*) but they are used independently from the *snrEval* parameters. The *ChannelControl* module only computes the distance in which nodes might potentially interfere with each other, i.e. in which *Omnet connections* are set up. The user can define up to what signal strength received power levels should be neglected via the signal attenuation threshold (*sat*), i.e. every signal that is weaker than *sat* is neglected. As a conclusion the same power- and frequency-values should be used for *snrEval*- and *ChannelControl*-parameters. If different transmitter power levels are used, the maximum power level has to be used for *pMax*. For more information on the *ChannelControl* module and connections in the MF see Section 7.2.

5.1 SnrEval

The *SnrEval* module simulates a transmission delay for all received messages and also calculates SNR information. The *BasicSnrEval* does not account for propagation delay. The SNR information is stored in a *SnrList*. Each *SnrList* entry contains a timestamp and a SNR value for this timestamp. The basic functions for *SnrEval* modules differ slightly from the ones defined in Section 4.1. *handleLowerMsg()* is subdivided into *handleLowerMsgStart()* and *handleLowerMsgEnd()*. Additionally we defined a *bufferMsg()* and an *unbufferMsg()* function.

Right after a message is received *handleLowerMsgStart()* is called. In this function a *SnrList* should be created to hold the SNR information for this frame and an initial entry should be added. Additionally the SNR information of all other messages in the receive buffer should be updated if desired. Afterwards the message is buffered (function *bufferMsg()*) to simulate a transmission delay. During this time other messages may arrive which would interfere with the buffered message and thus may result in additional *SnrList* entries to indicate a change in SNR for this message. After the transmission is complete (i.e. the message is completely received) *unbufferMsg()* un-buffers the message. *handleLowerMsgEnd()* is called right before the message is passed up to the *Decider* module. Here the message should be deleted from the receive buffer and the *SnrList* containing the calculated SNR values should be passed as an argument to the *sendUp()* function. The *sendUp()* function

There are several ways to implement *SnrEval* modules from only calculating one (average) SNR per message at the beginning of the reception to recalculating the SNR every time an additional message arrives resulting in a whole list of SNR values. We believe that our concept enables all these different models without being too complex but at the same time being sophisticated enough to also support complex models.

5.2 Decider

The *Decider* module only processes messages coming from the channel (i.e. from lower layers). Messages coming from upper layers bypass the *Decider* module and are directly handed to the *SnrEval* module. Decisions about bit error or lost messages only have to be made about messages coming from the channel. Consequently there is no need to process messages coming from upper layers in the *Decider* module.

The *Decider* module takes the *SnrList* created by the *SnrEval* module and translates the SNR values to bit errors. The simplest possible implementation would be to compare the SNR values against a SNR threshold. If at least one of the SNR values contained in the *SnrList* exceeds the SNR threshold the message is dropped due to bit errors. There are of course much more complex implementations possible as well.

As mentioned already earlier the *Decider* would also be the place to implement error detection and / or correction codes.

5.3 P2PPhyLayer

The big advantage of the *P2PPhyLayer* is that it is a lot faster than the subdivision into *SnrEval* and *Decider* modules. The price for the speed is that one cannot simulate sophisticated interference models and medium access techniques anymore.

P2PPhyLayer only takes a simple bit error probability *pBit* (usually from omnetpp.ini). This bit error probability covers all kinds of possible bit errors and messages losses. It thus also accounts for message losses due to collisions.

Consequently complex medium access techniques and interference models are not needed anymore. The advantage is that messages can be sent directly to the desired next hop and do not need to be broadcasted to all connected neighbors. This saves a lot of message duplication, sending and processing.

6 Using the Blackboard

When you evaluate the performance of a protocol, you need information on internal state changes of your protocol, maybe even from protocols that you use. You could monitor these changes using e.g. vector files from within you protocol and remove these monitors once you are done. Another way is to use a blackboard. The state changes are published on it, and the monitors subscribe to these values. This allows other researchers to tap your protocol for performance evaluation, while imposing nearly no runtime penalty when the information is not needed.

Maybe even more importantly, the blackboard allows you to exchange information between layers, without passing pointers to the modules around. Some items might not only be interesting for the layer they are created/changed in. The physical layer for example (*snrEval* in the MF) can sense whether a channel is busy or not. If the MAC protocol is based on carrier sense it needs the information the physical layer has. The *Blackboard* is a module where the corresponding information can be published and then is accessible for any module interested in it.

The *BasicModule* provides everything necessary to interact with the blackboard. It is derived from *ImNotifiable* – a pure virtual base class that allows the blackboard to inform your module of changes – and contains a pointer named *bb* to the blackboard module.

6.1 Methods for the subscriber

Subscribing to a parameter If you want to subscribe to a parameter, the best place to do so is the initialize function in stage 0. Suppose for example that you want to subscribe to changes of an item of the class *SomeBBItem*. When this item is changed, you have to be able to interpret its contents – and the way you can be sure that you can interpret the data is to know its type (or class in C++). Such items can be signal strength indicators, active radio channels, routing table entries, reasons why packets where lost and so forth, we will talk more about it in a separate section.

```
void YourClass::initialize(int stage) {
    BaseClass::initialize(stage);
    if (stage == 0){
        SomeBBItem item;
        catItem = bb->subscribe(this, &item, -1);
        ...
    }
    else if(stage == 1) {
   ...
```

In case you want to subscribe to a parameter, i.e. you want to be informed each time the content/value of that parameter changes, you have to call the *Blackboard* function *subscribe()*. You have to include a pointer to your module, a pointer to an object of class *SomeBBItem*, and a scope. This function returns an integer that is unique for the class *SomeBBItem*. The section 6.2 shows how to use this parameter. The Blackboard uses the *this* pointer to notify the module of published changes. The object pointer *& item* helps the *Blackboard* to learn something about the changes that you are subscribing. The *Blackboard* uses it to establish the connection between the *catItem* and the *&item*. The last parameter of the function determines a scope, it needs a more detailed explanation. A change can be published by several modules within a host. Consider the RSSI and a host with more than one network card. Each of these cards will publish a separate RSSI – but it will be of the same class and can hence not be distinguished. However, the meaning of the RSSI is constrained to one card (its scope): it makes no sense for a MAC protocol of one card to take the RSSI of the other card into consideration. The solution is to include the scope into the publication and let the *Blackboard* do some filtering. In the example code the subscriber uses -1 as the scope – a wildcard that subscribes him to all changes published by any module. If you want to subscribe to some specific stuff, you must make sure that the scope (for network cards usually the module id of the card) under which the parameter is published, matches the subscribed scope.

Un-subscribing from a parameter Sometimes parameter changes make only sometimes sense. For example, the host may decide to go to sleep. In this case it may be reasonable to unsubscribe:

```
bb->unsubscribe(this, catItem);
```

Your module will not get notifications for this parameter anymore.

6.2 Getting informed

Let us assume that you subscribed to a value and there is a valid publisher. The publisher informs the Blackboard of a change, which in turn calls the *receiveBBItem* method that your class inherited from the abstract *ImNotifiable* base class.

```
void YourClass::receiveBBItem(int category, \
const BBItem *details, int scopeModuleId) {
  // in case you want to handle messages here
  Enter_Method_Silent();
  // in case not you but your base class subscribed:
  BaseClass::receiveBBItem(category, details, scopeModuleId);
  // react on the stuff that you subscribed
  if(category == catItem) {
      someBBItemPtr =
            static_cast<const SomeBBItem *>(details);
      // do something
  }
}
```

The parameters of this function have already been explained in section 6.1. The first is the category, the integer that the subscribe function returned, the second is a pointer to an object of the class that you subscribed to and the third is the scope of this parameter. You should place the *Enter_Method* or *Enter_Method_Silent* macros at the beginning. This allows you to schedule or cancel messages, besides doing some animation. You should also inform the base class. This pattern is probably familiar from the initialize function. Now your base class will be informed about all changes it subscribed (and some of them are not interesting to *YourClass*) and the changes that *YourClass* subscribed. This has two implications:

if you forget this line, the simulation may stop completely or behave strange. Secondly, *YourClass* must gracefully handle any items that it did not subscribe. To support you in that task, the item that you receive from the Blackboard is read-only.

In the next step, the published item is handled. It is easy to determine its type: just check (may be in a switch statement) the category and cast it to the right class. Since the association between a class and its category is fixed, a static cast is safe and fast. Now *YourClass* can interpret the content and do something about it.

6.3 Methods for the publisher

Parameter You can only publish objects, since in C++ classes carry the semantic meaning. An integer can be used for everything: counting missed lunches, counting stars, or packets. If you would subscribe to an integer – how can your class know how to interpret it? This is where classes help you:

```
class  MissedLunches : public BBItem {
    BBITEM_METAINFO(BBItem);
protected:
    int counter;
public:
    double getMissedLunches () const {
        return counter;
    }
    void setMissedLunches(int c) {
        counter = c;
    }
    void addMissedLunch() {
       counter++;
    }

    MissedLunches(int c=0) : BBItem(), counter(c) {
       // constructor
    };
    std::string info() const { // for inspection
        std::ostringstream ost;
        ost << " You missed  " << counter << " lunches.";
        return ost.str();
    }
};
```

This associates the meaning “missed lunch” with an integer. At least, this is how most humans would interpret the name of the class. The class is derived directly from BBItem (which is derived from cPolymorphic), the base class for all items that can be published on the blackboard.

For advanced users: Of course it could also refine a class (say MissedMeals) and have a different parent class. The call to `BBITEM_METAINFO(BBItem);` helps the Blackboard to track the inheritance tree. If MissedLunches would have been derived from MissedMeals this should read `BBITEM_METAINFO(MissedMeals);`. This trick allows the Blackboard to deliver MissedLunches to all subscribers of MissedMeals.

After this you simply go ahead with a standard C++ class definition, and do what you like. The info function is optional, but helpful with debugging.

Publishing Ok, now that the parameter is defined, let us have a look how to publish it. First, some initialization is necessary. Let us assume that missedLunch is an object of class MissedLunches, and a member variable of YourPublisher.

```
void YourPublisher::initialize(int stage) {
    BasicModule::initialize(stage);
    if(stage == 0) {
        // initialize it
        missedLunch.setCounter(0);
        // get the category
        missedLunchCat = bb->getCategory(&missedLunch);
    }
    else if(stage == 1) {
        bb->publishBBItem(missedLunchCat, \
          &missedLunch, parentModule()->id());
    }
}
```

You should initialize missedLunches properly. The next step is to figure the category out, or put differently establish the connection between a category integer and a class. Now you are all set to publish it. We recommend that you publish your data in stage 1, this allows all subscribers to initialize the copies they might have. Publishing is a simple call to `bb->publishBBItem`. The first parameter is the category, the second a pointer to the published object that carries all the information that you want to publish, and the third is the scope. `parentModule()->id()` may not be useful as a scope in your case. There is no default or wildcard – you have to think about a reasonable and easy to understand scope.

Actually, we already covered how to publish a change when the simulation runs. Let us assume that YourPublisher tracks missed lunches somehow:

```
void YourPublisher::handleMissedLunch() {
        // update information
        missedLunch.addMissedLunch();
        // publish it.
        bb->publishBBItem(missedLunchCat, \
           &missedLunch, parentModule()->id());
}
```

YourPublisher simply has to update the information, and call the publish function again. That's it.

7 Mobility Modules

7.1 The Mobility Architecture

This section describes the mobility architecture used in the MF. First the general concept is described in Section 7.2. Section 7.3 describes how to create won mobility modules.

There are two mayor questions to consider for a mobility architecture in a simulation framework. The first question is where to process mobility information and and how to handle the movements of *Hosts*. Where and how to dynamically handle connections in an efficient way is the second decision to make.

We decided to handle mobility in a distributed manner locally in every *Host* module. Decisions how and where to move neither affect other *Hosts* nor do they require global knowledge. Connection management is handled centrally by one central controller. In order to set up and tear down connections the distances between *Hosts* have to be calculated for which we need the global knowledge of the position of all *Hosts*.

The core component of our mobility architecture is the global *ChannelControl* module together with an independent *MobilityModule* in each *Host* module (Fig. 5). *ChannelControl* handles all connection related things whereas the *MobilityModules* have two main tasks: The first task is to handle the movements of the *Host*. This can be done using various different mobility models (such as Manhattan Mobility or Brownian Motion). Second the *MobilityModule* communicates the location changes of the *Host* to the *ChannelControl* module. *ChannelControl* then updates all connection for this *Host*. The functionality of the *MobilityModules* is further described in Section 7.3.

7.2 ChannelControl

The *ChannelControl* module is responsible for establishing communication channels between *Host* modules that are within communication distance and tearing down these connections once they loose connectivity again. The loss of connectivity can be due to mobility (i.e. the *Hosts* move too far apart) or due to a change in transmission power or a crashed *Host* etc. We decided to keep the concept of links between *Host* modules, as opposed to direct message passing, since visible communication paths are an important source of (debugging) information in early development stages.

Unfortunately, in OMNeT++ distinct links between modules require at least two gate objects for each module, one in- and one out- gate (and for each submodule as well). For our MF the minimal number of gates per link is six since the *Nic* module is embedded within the *Host* module and is itself subdivided into an *SnrEval*, a *Decider* module and a *Mac* module. (see Section 4.3). To make sure to have enough gates even in the worst case scenario (all *Hosts* are directly connected), each *Host* module needs at least two pairs of gates for every single *Host* module in the network. Assuming n *Hosts* in the network one *Host* would need $6(n-1)$ gates or $3(n-1)$ gate pairs, leading to $6n(n-1) \approx 6n^2$ gates in the whole network.

A more memory-efficient approach is to create gates dynamically which is the way we decided to go. Gates are not allocated in bulk upon initialization of the network but created dynamically upon demand. Each *Host* module maintains two lists one for the free in-gates and one for the free out-gates. Once *ChannelControl* wants to establish a link between two *Hosts*, it first checks the gate lists in both *Hosts* whether free gates are available and only if no free gate was found a new one is created. Upon link break *ChannelControl* tears down the connection and adds the newly freed gates to the corresponding gate list. With this approach we minimize the memory needed without increasing the computational overhead to create and destroy gates too much.

In wireless network simulations not only the fact whether two hosts are connected (i.e. can communicate with each other) is important but also the fact whether two hosts can interfere with each other. That is why the term *connection* gets a slightly different meaning for our MF. Upon initialization, the *Channel-Control* module determines the maximum interference distance based on global network parameters such as the carrier frequency of the channel, the maximal possible sending power and other propagation specific parameters. The maximal interference distance is a conservative bound on the maximal distance at which a *Host* can still possibly disturb the communication of a neighbor, i.e. all *Hosts* further away will not recognize the sending signal at all. Please note that the maximal interference distance does neither specify the maximal distance at which messages can be (correctly) received nor does it specify the range at which *Hosts* definitely can receive some signal (even if it is only noise). Single *Hosts* in the network may have sending powers that are much less the maximum power specified and thus cannot reach a *Host* they are (theoretically) connected to. The maximum interference distance is just a theoretical means of reducing the computational overhead of our MF.

Based on the maximal interference distance *ChannelControl* calculates the connections between all *Hosts* upon initialization of the network and updates the connections every time a *Host* moves. Updating connections between *Hosts* is a computationally expensive operation. Calculating the distance between every pair of n *Hosts* in a network has a complexity of $\mathrm{O}(n^2)$. While this approach may not be the most efficient way to go, it is sufficient for the current version of our MF but could also be enhanced in case we experience performance problems.

7.3 Implementing Mobility Models

There is also the possibility to implement new mobility models. To do so, you have to derive the class for your new mobility module from *BasicMobility*. *BasicMobility* provides the *getRandomPosition()* function which selects a random starting position if no position is specified in the *omnetpp.ini* file. If you want to use another way of getting a staring position you can redefine this function.

Apart from that you have to do the following: At the times you want your host to move you have to send yourself a self-message and redefine the *handleSelfMsg()* function in which a new position should be determined. The position has to be of the type Coord (see API reference). As a final action in this function the *updatePosition()* function has to be called. It automatically updates the animated OMNeT++ GUI and furthermore writes the new position to the *Blackboard*. For further details take a look at the API reference of *BasicMobility*. As an example mobility implementation you can also have a look at the *ConstSpeedMobility* module in the *protocols/mobility* folder.

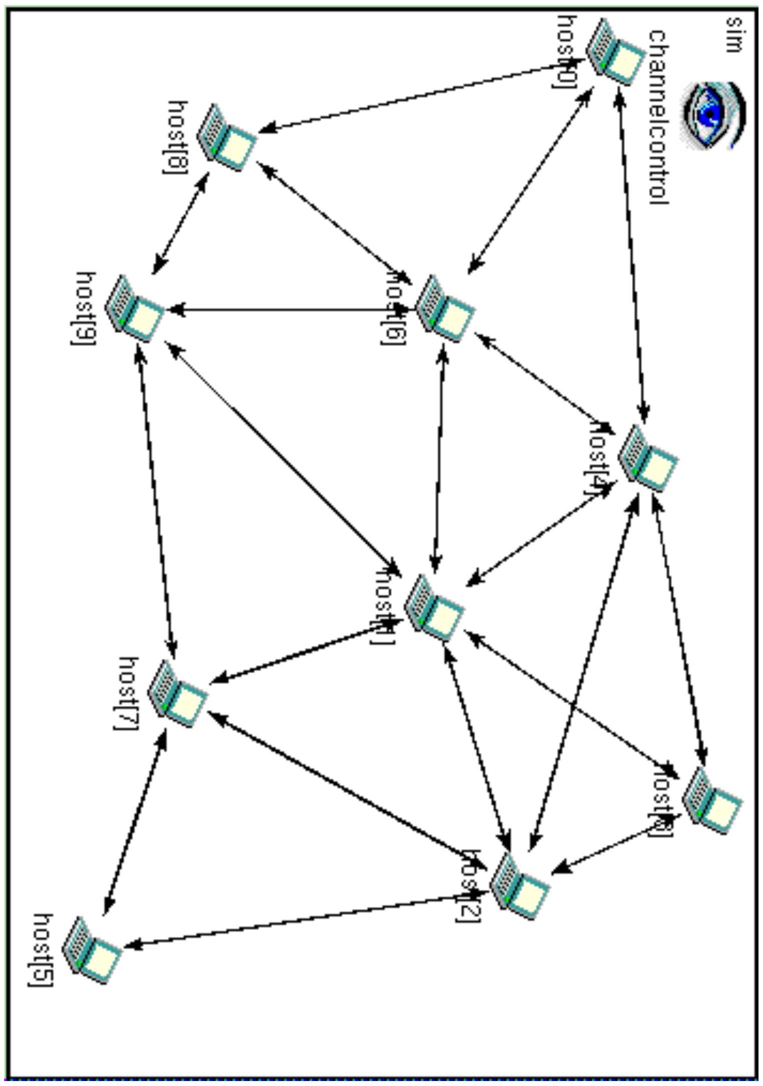

Figure 1: Simulation Setup with 10 nodes

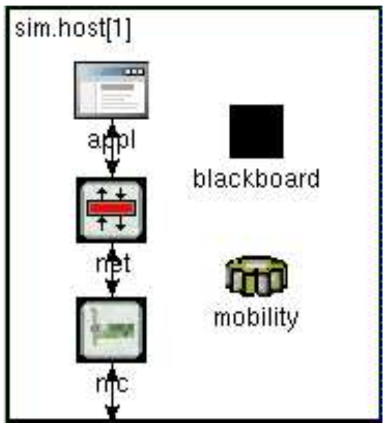

Figure 2: Structure of a Mobile Host

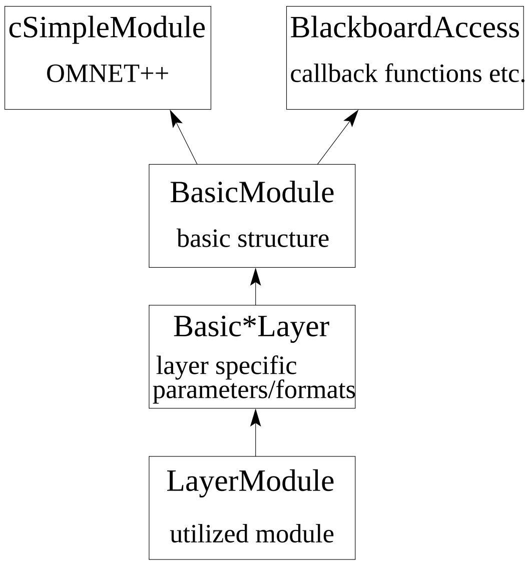

Figure 3: The general derivation structure

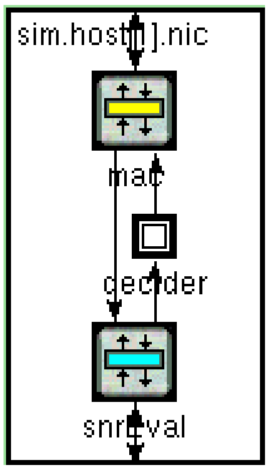

Figure 4: The structure of a nic module

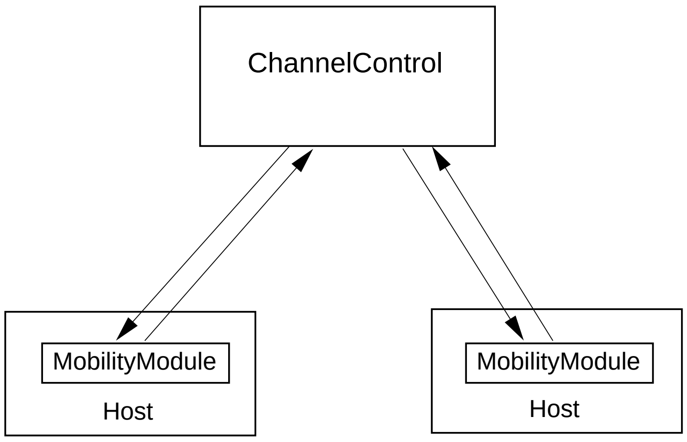

Figure 5: Mobility Architecture