## Existing models

During a simulation with WSNET, the entities on the xml file have an instance with a module. The modules are implementations of network models (radio propagation, interferences, mobility, MAC, application, etc). There are a certain number of models included in the online version of WSNET. The following table describes the models already implemented:

Table 1. Model in WSNet.

|  |  |
| --- | --- |
| **Layers** | **Implemented models** |
| Application | dummy, epidemic, Hello protocol, demo, CBR, CBR\_V2, B\_MAC\_app\_sample, dynamic |
| Routing | filestatic, geostatic, greedy, AODV, Directed Diffusion |
| MAC | Ideal MAC, IEEE 802.15.4 Beacon-less (868MHz - BPSK, 902MHz - BPSK, 2.4GHz - OQPSK) , IEEE 802.11 DCF, B-MAC, X-MAC |
| Transceiver | Half1d, Half1d\_wifi, 802.15.4 (868MHz - BPSK, 902MHz - BPSK, 2.4GHz - OQPSK), radio\_half\_duplex\_spectrum |
| Modulation | BPSK, classic\_OQPSK, OQPSK, MQAM, STEP, FSK, NONE |
| Interface | Omnidirectional |
| PHY | rf\_signal\_adjacent\_band |
| Spectrum | Multiband\_rf |
| Signal tracker | Pick\_first |
| Propagation | Pathloss: Free-space, 2-Ray Ground, ITU indoor model, pathloss\_wiplan, range, none, filestati, cost231, mixed  Shadowing: Lognormal, none, mixed  Fading : Rayleigh, Nakagami, None, mixed |
| Interferences (3.1) | None, orthogonal, factor, factor\_multichannel |
| Interference (4.0) | None, rf\_signal |
| Intermodulation | Intermodulation, none |
| noise | white |
| Mobility | Static, filestatic, Billiard, Torus Central, Torus Plane, Teleport, dummy, static\_wiplan, position\_inside\_room |
| Energy | Dummy, linear |
| Global\_map | Basic |
| Map | Dummy, indoor |
| Monitor | Dummy\_monitor |
| Physical | dummy |
| Sensor | dummy |
| Link | Dummy, indoor |
| Error | none |
| Coding | none |

### Generic Template

The first comments are a description of the module

/\*\*

\* \file FILE\_NAME.c

\* \brief BRIEF DESCRIPTION OF THE MODEL

\* \author AUTHOR NAMES

\* \date DATE

\* \version 1.0

\*\*/

Here we call the libraries that are necessary for all implementation. For example:

#include <include/modelutils.h>

#include <math.h>

#include <stdio.h>

#include "FILE\_NAME.h"

Modelutils library is always present in the implemented modules. You can also include other libraries needed for your model.

Some macro can be created using #define, which is the association of an identifier or parameterized identifier with a token string. After the macro is defined, the compiler can substitute the token string for each occurrence of the identifier in the source file.

#define

...

#endif

After we create some structures to manipulate the data. For example we can use nodedata for local variables and classdata for global variables.

struct nodedata {

int NODE\_LOCAL\_VARIABLE;

};

struct classdata {

int CLASS\_GLOBAL\_VARIABLE;

};

The data structure describes the module (authors, version, etc). It is necessary to call the model with the xml files. Make sure there is only one model\_t model structure in the code. You can also create more structures for your model as another C program.

model\_t model = { /\* Model description \*/

"MODULE DESCRIPTION",/\* one line description \*/

"AUTHORS",/\* author \*/

"0.1",/\* version \*/

MODELTYPE\_XX, /\* model type \*/

{NULL, 0} /\* list of available measures and number of measures \*/

};

In C++ implementation, the model structure is defined as follow:

model\_t model = {

(char\*) "MODULE DESCRIPTION",

(char\*)"AUTHORS ",

(char\*)"0.1",

MODELTYPE\_XX

};

The field "type" have to take one of the following values:

* MODELTYPE\_UNKNOW
* MODELTYPE\_PATHLOSS
* MODELTYPE\_SHADOWING
* MODELTYPE\_FADING
* MODELTYPE\_INTERFERENCES
* MODELTYPE\_NOISE
* MODELTYPE\_MODULATIONS
* MODELTYPE\_INTERMODULATION
* MODELTYPE\_MOBILITY
* MODELTYPE\_INTERFACE
* MODELTYPE\_TRANSCEIVER
* MODELTYPE\_MAC
* MODELTYPE\_ROUTING
* MODELTYPE\_APPLICATION
* MODELTYPE\_ENERGY
* MODELTYPE\_MONITOR
* MODELTYPE\_PHYSICAL
* MODEL\_TYPE\_SENSOR
* MODELTYPE\_MAP
* MODELTYPE\_GLOBAL\_MAP
* MODELTYPE\_LINK
* MODELTYPE\_PHY
* MODELTYPE\_ERROR
* MODELTYPE\_CODING
* MODELTYPE\_INTERFERENCE
* MODELTYPE\_SIGNAL\_TRACKER
* MODELTYPE\_SPECTRUM

#### Specific C implementation models

The init function is necessary to initialize the entity and the global entity parameters (values taken from the xml config file). Destroy is a function called at the end of the simulation to destroy the entity.

int init(call\_t \*to, void \*params) {

struct classdata \*classdata = malloc(sizeof(struct classdata));

param\_t \*param;

/\* get parameters \*/

list\_init\_traverse(params);

while ((param = (param\_t \*) list\_traverse(params)) != NULL) {

if (!strcmp(param->key, CLASS\_GLOCAL\_VARIABLE ")) {

if (get\_param\_double(param->value, &( classdata->CLASS\_GLOBAL\_VARIABLE))) {

goto error;

}

}

}

set\_class\_private\_data(to, classdata);

return 0;

error:

free(classdata);

return -1;}

int destroy(call\_t \*to) {

free(get\_class\_private\_data(to));

return 0;

}

Bind function is called to create the module, to allocate memory and initialize the parameters and variables specific to the node (local). Unbind function is called for the destruction of the module to free memory and remove parameters and local variables.

int bind(call\_t \*to, void \*params) {

struct nodedata \*nodedata = malloc(sizeof(struct nodedata));

param\_t \*param;

It is possible to initialize the values of LOCAL\_VARIABLE directly

nodedata-> NODE\_LOCAL\_VARIABLE= 0;

call\_t to0 = {get\_class\_bindings\_down(to)->elts[0], to->object};

call\_t from0 = {to->class, to->object};

It is possible to initialize the values of LOCAL\_VARIABLE using the XML file

list\_init\_traverse(params);

while ((param = (param\_t \*) list\_traverse(params)) != NULL) {

if (!strcmp(param->key, " NODE\_LOCAL\_VARIABLE ")) {

if (get\_param\_integer(param->value, &(nodedata-> NODE\_LOCAL\_VARIABLE))) {

goto error;

}

}

set\_node\_private\_data(to, nodedata);

return 0;

error:

free(nodedata);

return -1;

}

int unbind(call\_t \*to) {

struct nodedata \*nodedata = get\_node\_private\_data(to);

Deallocate the memory using packet\_dealloc, list\_destroy and free function

free(nodedata);

return 0;

}

Bootstrap function is called at the start of simulation to install and create the first event.

int bootstrap(call\_t \*to) {

return 0;

}

Ioctl function is used to exchange information between independent models.

int ioctl(call\_t \*to, int option, void \*in, void \*\*out) {

return 0;

}

The framework of one module is divided into two parts:

* One part which is the same for all of the module type. In this part, the functions are linked in modelutils.
* One part specific to one type of module. In this part, the functions are linked in model\_handlers.

The following structures define the framework specific to one module type.

MODEL\_TYPE\_methods\_t methods = {};

Note that you don’t have to use these functions directly in your module implementation, except for a few of them. You just need to implement them and they will be used by the simulator.

All the MODEL\_TYPE\_methods\_t type are defined in kernel/definition/models.h

#### Specific C++ implementation

The framework of a C++ module is divided into three parts:

* One part which is the same for all of the C++ module type. Any new C++ model should implement at least the create\_object, destroy\_object and boostrap functions in XX\_model\_api.cc as detailed in section 1.5.3.10.7.

void \*create\_object(call\_t \*to, void \*params) {

(void) to;

(void) params;

void \*p = TO\_C(new MODEL\_NAME\_Model(X,Y,Z));

return p;

}

void destroy\_object(void \*object) {

delete TO\_CPP(object, MODEL\_NAME\_Model);

}

int bootstrap(call\_t \*to) {

(void) to;

return SUCCESSFUL;

}

* One part specific to one type of module. In this part, the functions are linked in model\_handlers. The following structures define the framework specific to one module type. MODEL\_TYPE \_methods\_t methods = {};
* The last part is the definition of the class itself by inheriting and implementing the abstract class as defined in kernel/definitions/models/XX.h. This is the real definition of the class and, as default in C++, you should implement all virtual methods of the model abstract base class.

This specific CPP implementation is exclusive for the following models:

* PHY
* Spectrum
* Interference
* Signal Tracker
* Coding
* Error

You can take a look at the implementation of such models in the kernel to have a better idea of how those are implemented.

#### Mixed - C module with C++ implementation

Once C share a large common subset with C++, in some cases we can use C++ to create a model defined as C module. However, the other way is not possible, as C does not allow the use of classes.

Thus, in order to use C++ to create a model defined as C module we need to perform as the following:

* Create a cpp file following the framework specific for C module.
* At the top of your main source file (extension .cc, i.e. in C++), just after the definition of model\_t model you should declare the API functions as extern C, this is necessary to avoid name mangling from C++, thus enabling the correct generation of those symbols (which will be further read on runtime by during setup of the simulation)

extern "C"{

int init(call\_t \*to, void \*params);

int destroy(call\_t \*to);

int bind(call\_t \*to, void \*params);

int unbind(call\_t \*to);

int bootstrap(call\_t \*to);

int ioctl(call\_t \*to, int option, void \*in, void \*\*out);

}

* Implement all that you wish to implement using C++ STL, classes and etc.

### Application

The application layer defines a standard set of services and software for different applications that translate the sensor data in an understandable form or send queries to obtain certain information.

The library name that can be called in the configuration XML file are:

* application\_dummy\_application
* application\_epidemic
* application\_hello
* application\_demo
* application\_cbr
* application\_cbr\_v2
* application\_bmac\_app\_sample
* application\_dynamic

#### Methods Template

They are the methods that should be implemented by an application model.

typedef struct \_application\_methods {

void (\*rx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

} application\_methods\_t;

#### Dummy

Library name: **application\_dummy\_application**  
  
Description:  This application model is a dummy model.

Global Class parameter:

Local Node parameter:

#### Epidemic

Library name: **application\_epidemic**  
  
Description:  This application model is an epidemic data dissemination protocol. This protocol operates as follows:

* A source node broadcast periodically a new data
* A sensor node which receives a new data (not received previously), broadcasts it again according to a given probability p, and stores it locally according to a given probability q.

This protocol can be seen as a probabilistic flooding algorithm. Each data will be defined by a sequence number and a source number.

Two types of nodes have to be defined: the source and the sensor.

This model adds automatically the application header:

* the origin address
* the source address
* Sequence number

Global Class parameter:

* **p:**probability to forward the received packet. Default value is "1".
* **q:**probability to store the received packet. Default value is "1".

Local Node parameter:

* **Type:**type of the node: 0 for the sensor and 1 for the source. Default value is "0" (sensor).
* **Period:**transmission period of the source. Default value is "1s".

#### Hello protocol

Library name: **application\_hello**  
  
Description:  This application model is a simple hello protocol. This protocol starts to elect the source node as the closest node to the center of the simulation area. Each node sends periodically a HELLO packet. When a HELLO packet is received, the node updates the global matrix of connectivity.

This module provides some graph functionalities:

* Number of neighbors
* Minimal and maximal distances to a neighbors
* If the graph is connex
* …

The option graph is used to record in a file, the average node degree, the position and the connectivity of each node.

This model adds automatically the application header:

* The ID of Hello transmission.
* Data payload configurable by size

Global Class parameter:

Local Node parameter:

* **Start:**HELLO transmission start offset. Default value is "0".
* **Period:**HELLO transmission frequency. Default value is "1s".
* **Size:**size of HELLO packet. Default value is "10"bytes.
* **Graph:** flag to save the graph. Default value is "0".

#### Demo

Library name: **application\_demo**  
  
Description:  This application model is a demo model i.e. an example of WSNet3 simulation, aimed to demonstrate the ability of WSNet3 to handle several communication mediums. It uses also monitor and tracer.

Demo consists in 7 nodes (could be changed), distributed over 2 mediums:

3 nodes communicated over water, 3 nodes over air, and 1 behaving as a gateway between them. Each node, except the gateway, sends alternatively a message containing its id. This message is directly delivered to the node connected to the same medium as the sender, and forwarded to the other nodes by the gateway.

Three types of nodes have to be defined: the gateway with the double interface, air sensor and water sensor.

A tracer is used to trace the number of RX and TX by node. A monitor displays a message on each node birth event

This model adds automatically the application header:

* the source address

Global Class parameter:

Local Node parameter:

* **type:**type of the node: 0 for the sensor and 1 for the source. Default value is "0" (sensor).

#### CBR

Library name: **application\_cbr**  
  
Description:  This application model is a simple cbr application. Each node sends periodically a data packet. The data transmission starts at a random time in [start, start+period]. When a data packet is received, the node deletes the packet.

The destination is either a geographical destination or/and a node address. Both information are passed to the routing layer. Which will be considered depends on the underlying routing protocol e.g. routing\_greedy will consider the geographical position whereas routing\_filestatic will consider the destination address.

This model adds automatically the application header:

* Data payload configurable by size

Global Class parameter:

Local Node parameter:

* **start:**data transmission start offset. Default value is "0".
* **period:**data transmission period. Default value is "1s".
* **destination:**destination address. Default value is "random".
* **destination-x:**x coordinate of the geographical destination. Default value is "random".
* **destination-y:**y coordinate of the geographical destination. Default value is "random".
* **destination-z:**z coordinate of the geographical destination. Default value is "random".
* **size:**size of the transmitted data packet in B. Default value is "1000"B.

#### CBR\_V2

Library name: **application\_cbr\_v2**  
  
Description:  This application model is a simple cbr application with timers. Each node sends periodically a data packet using its own timer. The data transmission starts at a random time in [start, start+period]. When a data packet is received, the node deletes the packet.

The destination is either a geographical destination or/and a node address. Both information are passed to the routing layer. Which will be considered depends on the underlying routing protocol e.g. routing\_greedy will consider the geographical position whereas routing\_filestatic will consider the destination address.

This model adds automatically the application header:

* the source address
* the sequence number
* the delay
* the X and Y coordinate of the source

Global Class parameter:

Local Node parameter:

* **start:**data transmission start offset. Default value is "0".
* **period:**data transmission period. Default value is "1s".
* **destination:**destination address. Default value is "random".
* **destination-x:**x coordinate of the geographical destination. Default value is "random".
* **destination-y:**y coordinate of the geographical destination. Default value is "random".
* **destination-z:**z coordinate of the geographical destination. Default value is "random".

#### B\_MAC\_app\_sample

Library name:  **application\_bmac\_app\_sample**  
  
Description:  This application model is a sample time-driven application to be used with the B-MAC protocol. It periodically sends packets with a 6-bytes payload which includes 2 randomly generated integers. The destination is a node address or the broadcast address (it thus cannot be used with a routing protocol that considers geographical position such as routing\_greedy). It also uses the B-MAC IOCTL interfaces to enable the link-layer acknowledgments.

This model adds automatically the 6-Bytes application header:

* Sequence number
* Dummy1
* Dummy2

Global Class parameter:

Local Node parameter:

* **start:**data transmission start offset (relative to the birth of the node). Default value is "0".
* **stop:**data transmission stop offset (relative to the birth of the node). A value of "0" means that the transmission never stops (until node's death). Default value is "0".
* **tx\_period:**data transmission period. Default value is "1s".
* **random\_start:**whether transmission is started randomly in the [start, start+tx\_period] interval. Default value is "0".
* **dst:**destination address. Default value is "-1" (broadcast address)

#### Dynamic

Library name: **application\_dynamic**  
  
Description:  This application model is a dynamic (event/time/query-driven) application

In the time driven application, each node sends periodically a data packet using its own timer. The data transmission starts at a random time in [start, start+period].

In the event driven application, a node sends immediately a data packet when it receives an event from IOCTL with an event value higher than a threshold.

In the query driven application, a node sends immediately a data packet when it receives a query. This query packet can be generated after a QUERY\_MSG in IOCTL.

When a data packet is received, the node deletes the packet.

This model adds automatically the application header with the sequence number, but its size is the size of app\_data payload and the size of the structure pkt\_payload.

Global Class parameter:

Local Node parameter:

* **start:**data transmission start offset (relative to the birth of the node). Default value is "0".
* **stop:**data transmission stop offset (relative to the birth of the node). A value of "0" means that the transmission never stops (until node's death). Default value is "0".
* **Scheme:** Data collection scheme. 0 is time-driven, 1 is event-driven, and 2 is query-driven. Default value is "0" i.e. time driven.
* **tx\_period:**data transmission period for the time-driven scheme. 0 means that no data will ever be sent. Default value is "0".
* **random\_start:**whether transmission is started randomly in the [start, start+tx\_period] interval. Default value is "0".
* **Threshold:** threshold value for the event-driven scheme. Default value is "0".
* **dst:**destination address. Default value is "-1" (broadcast address)
* **payload:** size of the payload. Default value is "0" Byte.

### Routing

The network layer takes care of routing the data, directing the process of selecting paths along which to send data in the network.

The library name that can be called in the configuration XML file are:

* Routing\_filestatic
* Routing\_geostatic
* Routing\_greedy
* Aodv
* Directdiffusion

#### Methods Template

They are the methods that should be implemented by a routing model.

typedef struct \_routing\_methods {

void (\*rx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

void (\*tx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

int (\*set\_header) (call\_t \*to, call\_t \*from, packet\_t \*packet, destination\_t \*dst);

int (\*get\_header\_size) (call\_t \*to, call\_t \*from);

int (\*get\_header\_real\_size)(call\_t \*to, call\_t \*from);

} routing\_methods\_t;

#### Filestatic

Library name: **routing\_filestatic**  
  
Description:  This routing model is a static routing imported where routes are imported from a file.

This model adds automatically the routing headers and updates the next hop destination from the routes table:

* Generic routing header
  + the source address
  + the destination address

Global Class parameter:

* **file:** file containing the routes. Default value is "routing.data".

Local Node parameter:

#### Geostatic

Library name: **routing\_geostatic**  
  
Description:  This routing model is a static geographic routing. At the initialization, the routing model finds the node in range for each node and fills the neighbors table. This neighbor discovery is directly based on the real node positions without communication transaction.

When the static neighbor table is complete, several strategies to select the next hop destination are available:

* The next hop is the nearest neighbor from the destination that is still alive
* The next hop is chosen randomly among the neighbors that are nearer from the destination than the current node. It is a random geographic routing. The next hop is updated every "value" time.

If the destination is different from the last one, or if the current next hop is dead, the routing model reinitialize the random counters (to force the selection of a new next\_hop).

This model adds automatically the routing headers with the next hop destination from the routes table:

* Generic routing header
  + the source address
  + the destination address
  + Number of hop left

Global Class parameter:

Local Node parameter:

* **hop:**hop limit. Default value is "32".
* **range:**range in which neighbours are selected. Default value is "1".
* **random:**Randomize the choice of the next hop. 0 means never (always take the nearest one from the destination), and a value >= 1 randomizes the next hop every "value" time. Default value is "0".

#### Greedy

Library name: **routing\_greedy**  
  
Description:  This routing model is a greedy geographic routing. At the initialization, the routing model does not know the neighbors table. This neighbor discovery is based on a periodic Hello transmission in broadcast. The hello packet emission starts at a random instant in [start, start+period] and is repeated each period. When a hello packet is received, each node updates its neighbors table with its address, its location and its timestamp. A neighbor can expired and be deleted from the neighbors table after a configurable timeout.

The strategy to select the next hop destination:

* The next hop is the nearest neighbor from the destination that is still alive
* If the neighbor table is empty or a nearest neighbor does not exist, it returns NULL and the nest hop communication will not occur.

The generic routing header is :

* + the source address
  + the source position (x,y,z)
  + the destination address
  + the destination position (x,y,z)
  + Number of hop left
  + Type of the packet (Data or hello)

Global Class parameter:

Local Node parameter:

* **start:**hello transmission start offset. Default value is "0".
* **period:**hello transmission period. Default value is "1s".
* **timeout:**neighbor timeout. Default value is "2500ms".
* **hop:**hop limit. Default value is "32".

#### AODV

Library name:  **routing\_aodv**  
  
Description:

AODV (*Ad hoc On-Demand Distance Vector*) is a reactive routing protocol for mobile ad hoc networks (MANETs), where multi-hop routing paths are only established *on-demand*.

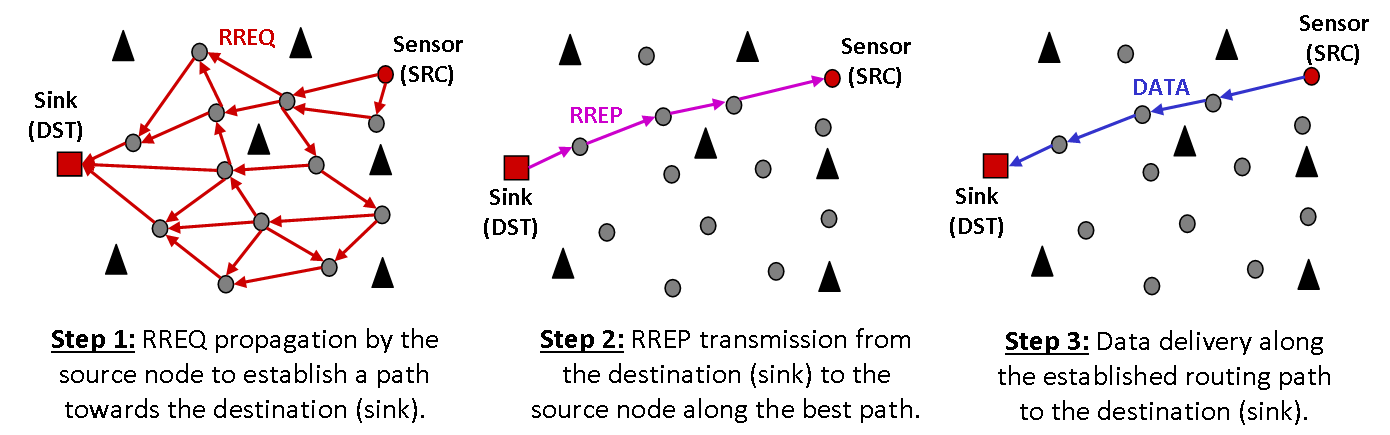


Figure 25. The AODV Routing Protocol.

The establishment of a multi-hop routing path between a source node and a destination is performed according to three main steps, as shown in Figure 25.

First, when a sensor node (*i.e.* the source) wants to transmit data packets to a given destination (*e.g.* the sink), it broadcasts a *Route Request* message (RREQ), as shown in Figure 25 (Step 1).

When the source’s neighbors receive the RREQ message, they have two choices: 1) if they know a route to the destination, a RREP (*Route Reply*) message is sent back to the source node using *unicast* routing method, as shown in Figure 25 (Step 2); otherwise 2) the RREQ message is rebroadcast again. In this latter case, the RREQ message keeps getting rebroadcast until reaching the destination, as shown in Figure 25 (Step 1), or until its life span is used up, for example, using a TTL (*Time-To-Live*) based approach.

Finally, as shown in Figure 25 (Step 3), when the source node receives a RREP message, it transmits the data packet along the established routing path using a *unicast* based routing method.

Global Class parameter:

Local Node parameter:

* **node\_type**: this defines the type of the wireless node. Possible values are: “0”, “1” or “2” for sensor, sink and anchor types, respectively. The default value is “0”.
* **sink\_id**: this defines the ID of the destination sink node. The default value is “-1”, i.e. the sink ID is unknown.
* **hello\_status**: this enables or disables the neighbor discovery protocol. Possible values are: “0” to disable the hello protocol and “1” to enable it. By default, this parameter is equal to “1”.
* **hello\_start**: this defines the startup time of the neighbor discovery protocol. By default, the hello protocol starts at the simulation startup (i.e. hello\_start = “0” second) and the time unit is nanosecond (ns) .
* **hello\_period**: this defines the periodicity of the hello packet transmission. By default, this parameter is equal to “1” second and the time unit is nanosecond (ns) .
* **hello\_nbr**: this defines the maximal number of hello packets to be transmitted. By default, this parameter is equal to “-1” (i.e. hello packets will be generated until the end of the simulation). Other possible input values are integers > 0.
* **hello\_timeout**: this defines the period of time after which a neighbor is removed from the local neighbor table if no hello packets were received. By default, this parameter is equal to “3” seconds and the time unit is nanosecond (ns)
* **rreq\_status**: this enables or disables the periodic generation and transmission of RREQ packets. Possible values are: “0” to disable the RREQ periodic generation and “1” to enable it. By default, this parameter is equal to “0”.
* **rreq\_nbr**: this defines the maximal number of RREQ packets to be transmitted by each node. By default, this parameter is equal to “-1” (i.e. RREQ packets will be generated until a RREP is received). Other possible input values are integers > 0.
* **rreq\_start**: this defines the startup time of the RREQ periodic generation. By default, the RREQ protocol starts at the simulation startup (i.e. rreq\_start = “0” second) and the time unit is nanosecond (ns)
* **rreq\_period**: this defines the periodicity of the RREQ packet transmission. By default, this parameter is equal to “10” seconds and the time unit is nanosecond (ns)
* **rssi\_smoothing1\_nbr**: this defines the required number of readings to smooth RSSI measurements for each link (or neighbor). By default, this parameter is equal to “1” (i.e. 1 reading to smooth RSSI measurements). Other possible input values are integers > 0.
* **rssi\_smoothing2\_nbr**: this defines the required number of readings to smooth RSSI measurements for each link (or neighbor). By default, this parameter is equal to “8” (i.e. 8 readings to smooth RSSI measurements). Other possible input values are integers > 0.
* **lqe\_w**: this defines the size of the history () for the computation of the link quality estimation. By default, this parameter is equal to “10”.
* **lqe\_threshold**: this defines the threshold for the selection of the most reliable radio links. By default, this parameter is equal to “0.9”.
* **hello\_packet\_real\_size**: this defines the real size of hello packets in bytes. By default, this parameter is equal to 13 bytes. Other possible input values are integers > 0.
* **rreq\_packet\_real\_size**: this defines the real size of RREQ packets in bytes. By default, this parameter is equal to 13 bytes. Other possible input values are integers > 0.
* **rrep\_packet\_real\_size**: this defines the real size of RREP packets in bytes. By default, this parameter is equal to 13 bytes. Other possible input values are integers > 0.

#### Directed Diffusion

Library name: **routing\_directdiffusion**   
  
Description:

Numerous routing techniques have been proposed for the particular context of wireless sensor networks (WSNs). A popular routing technique for wireless sensor networks with static sinks has been proposed, namely the Directed Diffusion approach. This module is based on the Directed Diffusion based routing approach, a popular routing technique that was specifically designed for data dissemination and collection in wireless sensor networks with static sinks.

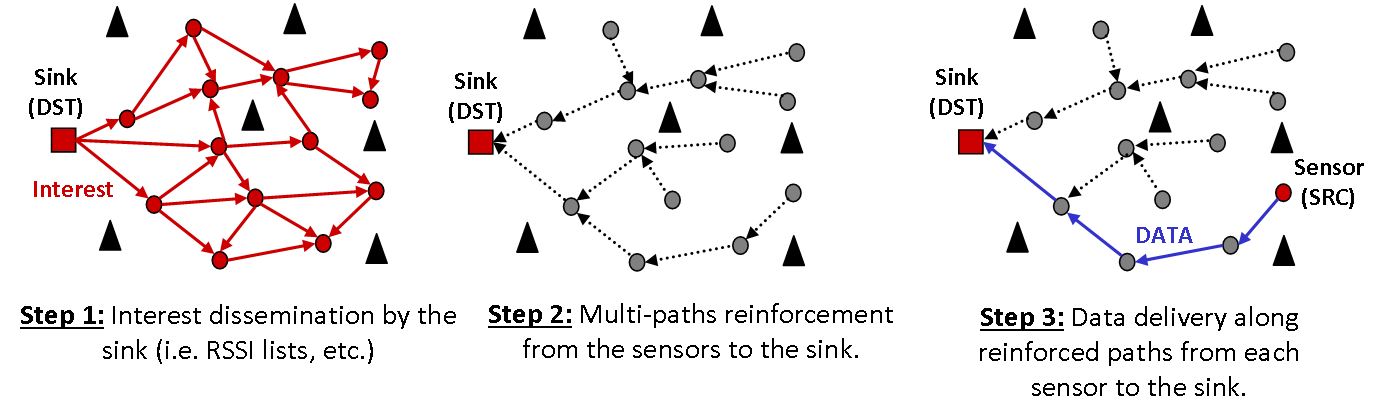


Figure 26. The Directed Diffusion based Routing Protocol.

In Directed Diffusion the sink node requests data from the sensors by broadcasting interests. An interest describes a task required to be done by the sensors (*e.g.* RSSI lists delivery at the sink, *etc.*).

Interest packets are broadcasted hop-by-hop through the network, as shown in Figure 26 (Step 1). As these packets are propagated throughout the network, gradients (or routes) are set up towards the requesting sink, as shown in Figure 26 (Step 2). At the end of these two steps (*cf.* Step 1 and 2 in Figure 26), all the sensors are aware of: 1) *the task to be done* (e.g. RSSI gathering and RSSI lists delivery to the requesting sink); and 2) *the next hop sensor* (or route) towards the requesting sink for the delivery of data packets, as shown in Figure 26 (Step 3).

It should be noted that the first step (cf. Step 1 in Figure 26) corresponds to the *data dissemination phase*, while the two latter steps (cf. Step 2 and 3 in Figure 26) are related to the *data collection phase*.

Global Class parameter:

Local Node parameter:

* **node\_type**: this defines the type of the wireless node. Possible values are: “0”, “1” or “2” for sensor, sink and anchor types, respectively. The default value is “0”.
* **hello\_status**: this enables or disables the neighbor discovery protocol. Possible values are: “0” to disable the hello protocol and “1” to enable it. By default, this parameter is equal to “1”.
* **hello\_start**: this defines the startup time of the neighbor discovery protocol. By default, the hello protocol starts at the simulation startup (i.e. hello\_start = “0” second) and the time unit is nanosecond (ns) .
* **hello\_period**: this defines the periodicity of the hello packet transmission. By default, this parameter is equal to “1” second and the time unit is nanosecond (ns) .
* **hello\_nbr**: this defines the maximal number of hello packets to be transmitted. By default, this parameter is equal to “-1” (i.e. hello packets will be generated until the end of the simulation). Other possible input values are integers > 0.
* **hello\_timeout**: this defines the period of time after which a neighbor is removed from the local neighbor table if no hello packets were received. By default, this parameter is equal to “3” seconds and the time unit is nanosecond (ns).
* **sink\_interest\_status**: this enables or disables the interest dissemination by the sink node. Possible values are: “0” to disable the interest dissemination and “1” to enable it. By default, this parameter is equal to “0”.
* **sink\_interest \_start**: this defines the startup time of the interest dissemination by the sink node. By default, the interest protocol starts at the simulation startup (i.e. interest\_start = “0” second) and the time unit is nanosecond (ns) .
* **sink\_interest\_period**: this defines the periodicity of the interest dissemination by the sink node. By default, this parameter is equal to “10” seconds and the time unit is nanosecond (ns).
* **sink\_interest\_nbr**: this defines the maximal number of interest packets to be transmitted by the sink. By default, this parameter is equal to “-1” (i.e. interest packets will be generated until the end of the simulation). Other possible input values are integers > 0.
* **sink\_interest\_data\_type**: this defines the type of the requested data by the sink. By default, this parameter is equal to “-1” (all data types are requested by the sink). Other possible input values are integers ≥ 0.
* **sink\_interest\_ttl**: this defines the TTL (time-to-live) of interest packets to be transmitted by the sink. By default, this parameter is equal to 100”. Other possible input values are integers > 0.
* **sink\_interest\_propagation\_backoff**: this defines the backoff time associated to the propagation of received interest packets. By default, this parameter is equal to “1” second and the time unit is nanosecond (ns) .
* **sink\_interest\_propagation\_probability**: this defines the probability to rebroadcast received interest packets. By default, this parameter is equal to “1”. Possible values are real numbers between 0.0 and 1.0.
* **lqe\_w**: this defines the size of the history () for the computation of the link quality estimation. By default, this parameter is equal to “10”.
* **lqe\_threshold**: this defines the threshold for the selection of the most reliable radio links. By default, this parameter is equal to “0.9”.
* **rssi\_smoothing1\_nbr**: this defines the required number of readings to smooth RSSI measurements for each link (or neighbor). By default, this parameter is equal to “1” (i.e. 1 reading to smooth RSSI measurements). Other possible input values are integers > 0.
* **rssi\_smoothing2\_nbr**: this defines the required number of readings to smooth RSSI measurements for each link (or neighbor). By default, this parameter is equal to “8” (i.e. 8 readings to smooth RSSI measurements). Other possible input values are integers > 0.
* **hello\_packet\_real\_size**: this defines the real size of hello packets in bytes. By default, this parameter is equal to 13 bytes. Other possible input values are integers > 0.
* **interest\_packet\_real\_size**: this defines the real size of Sink Interest packets in bytes. By default, this parameter is equal to 13 bytes. Other possible input values are integers > 0.

### MAC

The MAC or data link layer is responsible for the channel access technique by providing the multiplexing of data streams, data frame detection and medium access control.

The library name that can be called in the configuration XML file are:

* mac\_ideal\_mac
* mac\_802\_15\_4\_u\_csma\_ca\_2400\_oqpsk
* mac\_802\_15\_4\_u\_csma\_ca\_\_868\_bpsk
* mac\_802\_15\_4\_u\_csma\_ca\_\_902\_bpsk
* mac\_dcf\_802\_11
* mac\_bmac
* mac\_xmac

#### Methods Template

They are the methods that should be implemented by a MAC model.

typedef struct \_mac\_methods {

void (\*rx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

void (\*tx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

int (\*set\_header) (call\_t \*to, call\_t \*from, packet\_t \*packet, destination\_t \*dst);

int (\*get\_header\_size) (call\_t \*to, call\_t \*from);

int (\*get\_header\_real\_size)(call\_t \*to, call\_t \*from);

} mac\_methods\_t;

#### Ideal MAC

Library name: **mac\_idealmac**  
  
Description:  This model is an ideal MAC layer. It does not use the WSNet kernel and the appropriate medium models (no propagation, no interference, no collisions, etc.). This model distributes all packets to node in communication range. Thus all packets are received correctly by nodes within reach communication. A half-duplex option is possible.

This model also adds automatically the MAC layer header:

* 1 Byte for the source address
* 1 Byte for the destination address
* 1 Byte for the packet type

Global Class parameter:

**range:**the range of the unit-disk model. Default value is "10"m.

**bandwidth:**Data bandwidth. Default value is "15" Kbyte/s

Local Node parameter:

#### IEEE 802.15.4 Beacon-less (868MHz - BPSK, 902MHz - BPSK, 2.4GHz – OQPSK)

Library name: **mac\_802\_15\_4\_868\_bpsk\_u\_csma\_ca /mac\_802\_15\_4\_902\_bpsk\_u\_csma\_ca/ mac\_802\_15\_4\_2400\_oqbsk\_u\_csma\_ca**  
  
Description:  This model is an implementation of the IEEE 802.15.4 unslotted csma/ca protocol. This protocol is a listen before talk protocol based on a backoff procedure. The protocol parameters are given by the 802.15.4 standard. 3 configurations are available:

* **868MHz bpsk**
* **902MHz bpsk**
* **2400MHz bpsk**

When a packet is ready to be transmitted, the MAC module launches the backoff procedure of the unslotted CSMA/CA.

* It initializes the number of attempts to 0 and the backoff exponent BE to macMinBE.
* The node waits during a random period (Backoff delay) :
* At the end of this period, the node performs a CCA:
  + If the channel is free, the procedure is considered as successful
  + If the channel is busy, the node increments NB and BE and launches again the procedure i.e. the node waits during another random backoff delay (but longer than the previous one) before checking again the channel state. The number of backoff attempts is limited to max\_csma\_backoff.

The following figure detail the backoff procedure.

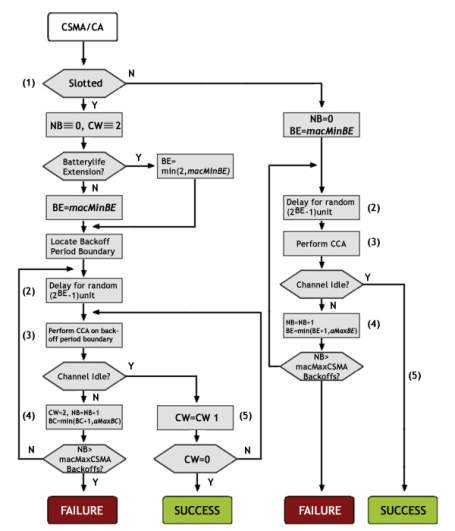


Figure 27. CSMA/CA algorithm

The MAC layer also manages the timing of communications:

* The duration of a backoff unit is 20 symbols i.e. 1 ms in BPSK\_868, 500µs in BPSK\_902 and 320 µs in OQPSK\_2400.
* The duration of an energy detection is 8 symbols i.e. 400 µs in BPSK\_868, 200µs in BPSK\_902 and 128 µs in OQPSK\_2400.
* The interframe duration macMinSIFSPeriod after a short packet (< 18 Bytes) is 12 symbols i.e. 600 µs in BPSK\_868, 300µs in BPSK\_902 and 192 µs in OQPSK\_2400.
* The interframe duration macMinLIFSPeriod after a long packet (> 18 Bytes) is 40 symbols i.e. 2 ms in BPSK\_868, 1 ms in BPSK\_902 and 640 µs in OQPSK\_2400.

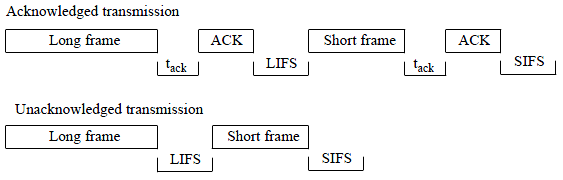


Figure 28. Interframe duration

Moreover, the MAC model defines acknowledgement and retransmission mechanisms. When a packet is not correctly acknowledged the MAC reuses the same packet for the next backoff procedure. The number of retransmission is limited to MAC\_max\_retries.

The following MAC state machine details the algorithm implementation:

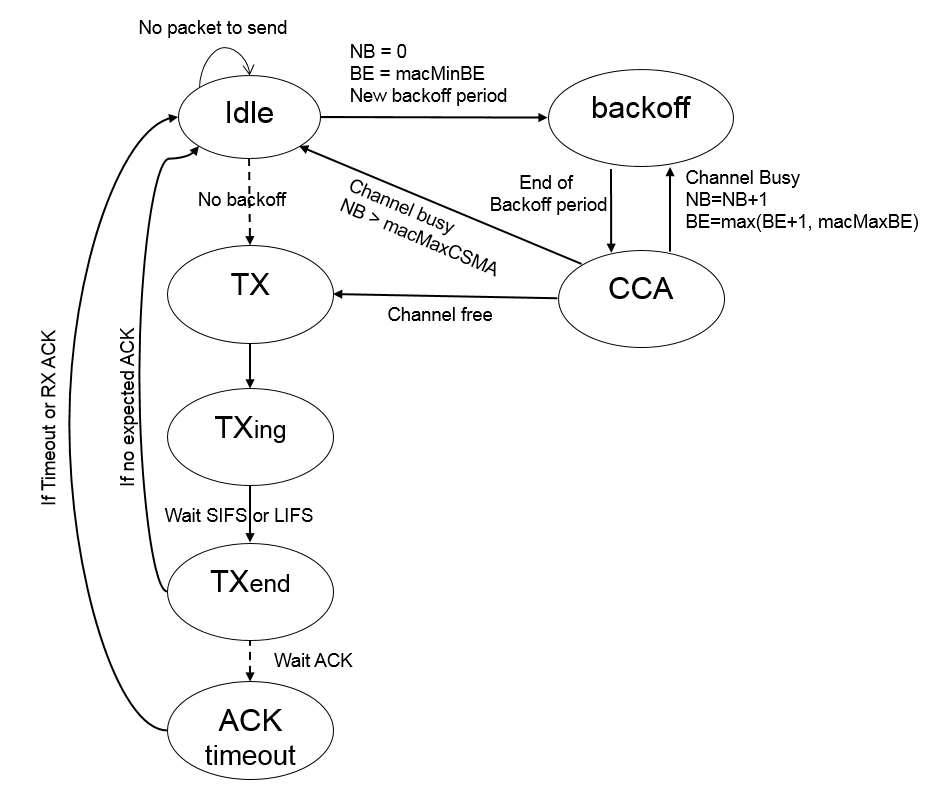


Figure 29. Unslotted CSMA/CA MAC State Machine

During the initialization, all the nodes start in IDLE state. If a node has a packet to send i.e. the packet buffer is not empty, it launches the backoff procedure and switches to backoff state.

At the end of the backoff period, it switches in CCA state during the duration of the energy detection. If the channel is busy, it goes back in backoff state or in idle state. If the procedure succeeds, it goes in TX state and then starts to transmit its packet.

If the packet does not need an acknowledgement, it can deallocate its packet and waits macMinSIFSPeriod or macMinLIFSPeriod before going back to Idle state in which it will check if another packet must be sent.

If the packet needs an acknowledgement, the node asks to the receiver to send back an acknowledgement packet and waits during a timeout. If an ACK is received, the node deallocates the packet and returns in IDLE state after macMinSIFSPeriod or macMinLIFSPeriod. If no ACK is received before the timeout, the retransmission procedure can launch again MaC\_max\_retries time the backoff procedure with the same packet.

In the bootstrap, the MAC layer wakes up the transceiver and switches it in reception mode.

IEEE 802.15.4 unslotted CSMA\CA is an always on mode. Then the transceiver must be in reception mode all the time except when it is transmitting. Thus after a transmission or a reception, the MAC switched the transceiver to RX mode.

This model also adds automatically the 2 MAC layer headers:

* Generic 802.15.4 header
  + 2 Bytes for the Frame control (frame type, flag for acknowledgement…)
  + 1 Byte for the sequence number
  + 2 Bytes for the CRC
* Data 802.15.4 header
  + 2 Bytes for the source address
  + 2 Bytes for the destination address
  + 2 Bytes for the PAN identifier

The generic header is also used for MAC control packets.

Global Class parameter:

Local Node parameter:

**cca:**whether the channel energy must be sensed for clear channel assessment. Default value is "1".

**cs:**whether the carrier sense must be checked for clear channel assessment. Default value is "1".

**cca-threshold:**carrier sense and/or energy threshold for clear channel assessment. Default value is "-82".

**max-csma-backoffs:**number of backoff attempts before the packet is dropped. Default value is "4", must be between 0 and 5**.**

**min-backoff-exponent:**minimum value for the backoff exponent. Default value is "3", must be superior or equal to 0.

**max-backoff-exponent:**maximum value for the backoff exponent. Default value is "5", must be between 3 and 8.

**cca-threshold:**This parameter defines the threshold on which we consider the channel busy. Default value depends on the band: -82 dBm for BPSK\_868 and BPSK\_902 and -75 dBm for OQPSK\_2400.

**acknowledgment:**This parameter indicates if the acknowledgement mechanism is activated.Default value is deactivated (0).

**MAC\_max\_retries:**Maximal number of retransmission attempts**.** Default value is "0"

#### IEEE 802.11 DCF

Library name: **mac\_dcf\_802\_11**

Description:  This model is an implementation of the IEEE 802.11 unslotted csma/ca protocol with RTS/CTS. This protocol is a listen before talk protocol based on a backoff procedure. The protocol parameters are given by the 802.11 standard.

When a packet is ready to be transmitted, the MAC module can directly transmit after a DIFS if the channel is free. Otherwise it launches the backoff procedure of the unslotted CSMA/CA.

* It initializes the number of attempts to 0 and the backoff exponent BE to macMinBE.
* The node waits during a random period (Backoff delay) :

The backoff timer is decremented by one during each idle time-slot within the contention periods.

The channel is expected to be idle if:

* + No transmissions are detected (physical carrier sensing) and
  + NAV (Network allocation vector) counter is zero (virtual carrier sensing). A station listens to the MAC headers of all the frames in the air, and updates the counter based on the frame length information in each MAC header. The NAV mechanism partly avoids the problems due to hidden nodes.
* A station starts transmitting when the timer reaches zero.
* If a contention was not successful for a station, it continues during a new contention period from the end value reached during the previous contention period.

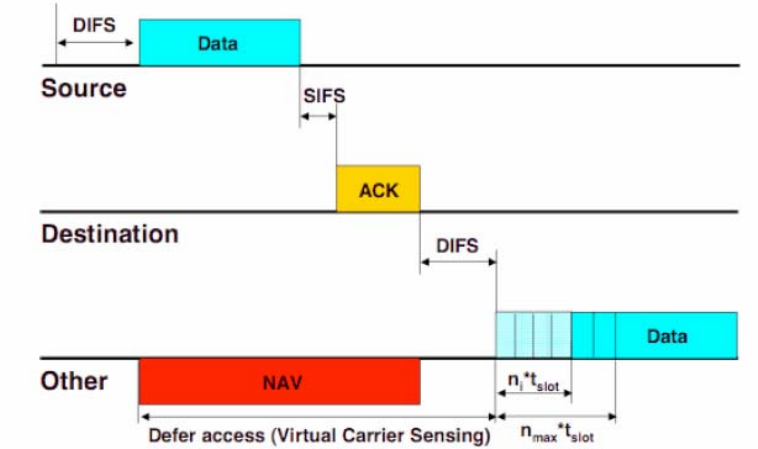


Figure 30. Transmission of a Data with acknowledgement in IEEE802.11

If the ACK is missing, the transmitting station increments the retry counter. It doubles the CW (up to a certain limit) and gets a new back-off timer value. When the maximum number of retransmissions is reached, the packet is discarded.

If the Data is larger than a threshold, a 4-way handshake (RTS, CTS, DATA, ACK) protocol is used.

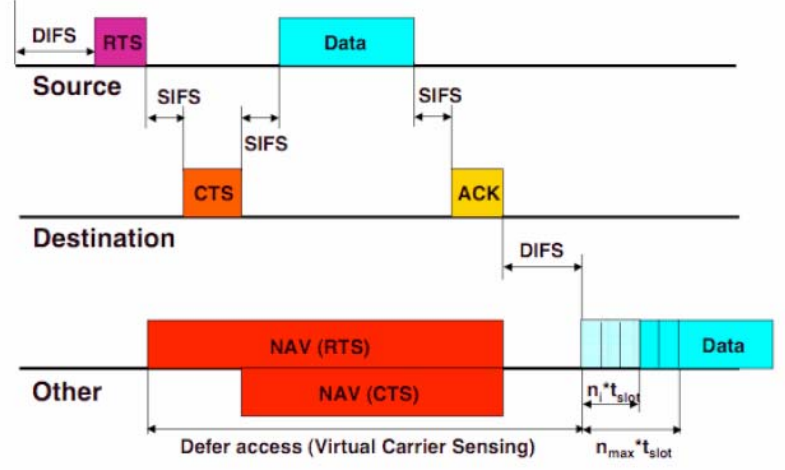


Figure 31. Transmission of a Data with ACK/RTS/CTS in IEEE802.11

When a sending station wants to transmit data, it first sends (through contention) an RTS (request to send) frame and waits for the destination to reply with a CTS (clear to send) frame. If CTS is detected, then data is transmitted and destination sends an ACK if the data was received completely and correctly. If CTS is not detected, the sending station goes back to compete for the media, in the CSMA/CA mode

All other stations that hear RTS or CTS would defer transmission in the duration indicated in RTS or CTS. Both RTS and CTS include the estimated transmission time in the MAC header, so stations which hear either of them, know when the channel is going to be free again, through the NAV mechanism.

RTS is not used for short packets, for which the cost of collision is lower. RTS packets of different stations may collide anyway. And the RTS/CTS mechanism would cause significant overhead for small packets.

The MAC layer also manages the timing of communications:

* The duration of a backoff unit is 20 µs.
* The interframe duration macMinSIFSPeriod is 10µs.
* The minimum delay before starting transmission after the channel is expected to be idle macMinDIFSPeriod is 50 µs.

In the bootstrap, the MAC layer wakes up the transceiver and switches it in reception mode.

IEEE 802.11 unslotted CSMA\CA is an always on mode. Then the transceiver must be in reception mode all the time except when it is transmitting. Thus after a transmission or a reception, the MAC switched the transceiver to RX mode.

This model defines several MAC layer headers:

* Generic 802.11 header is 10 bytes.
  + the source address
  + the destination address
  + the type
* RTS 802.11 header is 6 bytes. Thus the size of on RTS is 16 Bytes
  + The NAV
  + The size
* CTS 802.11 header is 0 byte. Thus the size of on CTS is 10 Bytes
  + The NAV
* DATA 802.11 header is 22 bytes. Thus the size of on DATA header is 32 Bytes
  + The NAV
  + The size
* ACK 802.11 header is empty. Thus the size of on ACK is 10 Bytes

Global Class parameter:

**retry:**number of transmission retries before a unicast packet is dropped. Default value is "7".

Local Node parameter:

**rts-threshold:**packet size threshold for the use of RTS/CTS handshake. Default value is 500B**.**

**cca:**whether the channel energy must be sensed for clear channel assessment. Default value is "1".

**cs:**whether the carrier sense must be checked for clear channel assessment. Default value is "1".

**cca-threshold:**carrier sense and/or energy threshold for clear channel assessment. Default value is "-74".

#### B-MAC

Library name: **mac\_bmac**  
  
Description:  This model is the Berkeley Media Access Control model. BMAC is a contention-based protocol, through Low Power Listening (LPL) ensuring power management. A node maintains a listening duty cycle divided by a specific time period called check interval. Node checks activity at each check interval when it wakes up. When activity is detected, node stays awake to receive incoming packet. But, if the medium is clear, node returns to sleep. Every transmission, to support LPL is preceded by a preamble till the check interval so that the intended receiver is transmission aware and receives incoming packet. This shifts load from receivers to senders, saving much energy during light traffic load.

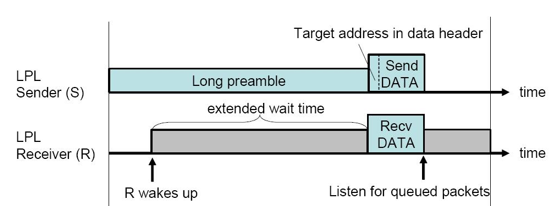


Figure 32. BMAC protocol

Several strategies are available from always on transceiver (LPL mode 0) to each 1600 ms (LPL mode 8).

When a packet is ready to be transmitted, the MAC module launches the backoff between 0 and 10 ms. At the end of this period, the node performs a CCA:

* + If the channel is free, the procedure is considered as successful
    - It starts to send some preamble packets
    - Then it sends a sync word packet
    - Finally it sends the data packet
  + If the channel is busy, it checks up to 5 times the state of the channel. If the channel is still busy, it can launch a new backoff. If the channel becomes free, it can complete the procedure.
  + When the data is sent, an ACK procedure can be launched.

This model defines a generic BMAC headers:

* the source address
* the destination address
* the type

Global Class parameter:

Local Node parameter:

**cca:**whether the channel energy and carrier sense must be checked for clear channel assessment. Default value is "1".

**busy-threshold:**carrier sense and energy threshold for clear channel assessment. Default value is "-74"dBm.

**ack:**whether link-layer acknowledgments are enabled for unicast packets. Default value is "1".

**max-retrans:**number of transmission retries before a unicast packet is dropped. Default value is "2".

**lpl-check**: low-power-listening sampling period and preamble length. If set to "0", the node is always on and does not send any preamble before the data. Default value is "100ms".

**init-back:**value of the initial backoff. Default value is a random number between 1 and 10ms.

**cong-back:**value of the congestion backoff. Default value is a random number between 1 and 10ms.

#### X-MAC

Library name: **mac\_xmac**  
  
Description:  This model is the XMAC model. X-MAC is a preamble sampling MAC protocol. It sends a serie of short preamble packets, each containing ID of target node: nodes receiving the preamble and that are not recipient of the ensuing data go back to sleep. A short pause between preamble packets allows recipient to send an early acknowledgement to receive the data sooner. Potential senders that overhears such ack and that wish to send to the same node can queue the packets directly after (after a random backoff) the data sent by the other transmitter

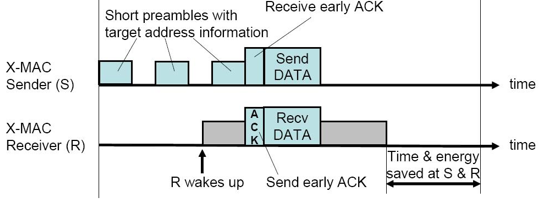


Figure 33. XMAC protocol

Several strategies are available from always on transceiver (LPL mode 0) to each 1600 ms (LPL mode 8).

When a packet is ready to be transmitted, the MAC module launches the backoff between 0 and 10 ms. At the end of this period, the node performs a CCA:

* + If the channel is free, the procedure is considered as successful
    - X-MAC divides the preamble into small packets sent consecutively on the medium. The entire preamble is a little longer than the LPL check interval. It adjusts preamble\_count to the length of a preamble word plus the pause between the preambles
    - If a node receives a preamble packet with its address inside, it sends a preamble ACK after an interframe space.
    - If a node sending a serie of short preamble receives a preamble ACK from the target node, it stops sending preamble and directly starts to send its data packet. If it has finished to send its serie of preamble packet without receiving an preamble ACK, it sends the data packet.
  + If the channel is busy, it checks up to 5 times the state of the channel. If the channel is still busy, it can launch a new backoff. If the channel becomes free, it can complete the procedure.
  + When the data is sent, an ACK procedure can be launched.

If the packet is a broadcast one, it can be sent directly after the whole preamble (no ACK were expected). If it a unicast packet, a Preamble ACK was expected but not received. We however still send the packet as the Preamble ACK was maybe lost.

This model defines a generic BMAC headers:

* the source address
* the destination address
* the type

Global Class parameter:

Local Node parameter:

**cca:**whether the channel energy and carrier sense must be checked for clear channel assessment. Default value is "1".

**busy-threshold:**carrier sense and energy threshold for clear channel assessment. Default value is "-74"dBm.

**ack:**whether link-layer acknowledgments are enabled for unicast packets. Default value is "1".

**max-retrans:**number of transmission retries before a unicast packet is dropped. Default value is "2".

**lpl-check**: low-power-listening sampling period and preamble length. If set to "0", the node is always on and does not send any preamble before the data. Default value is "100ms".

**init-back:**value of the initial backoff. Default value is a random number between 1 and 10ms.

**cong-back:**value of the congestion backoff. Default value is a random number between 1 and 10ms.

### Transceiver

The physical layer can provide an interface to transmit a stream of bits over physical medium. It is responsible for frequency selection, carrier frequency generation, signal detection, modulation and data encryption.

The transceiver module models the radio state machine and its different modes:

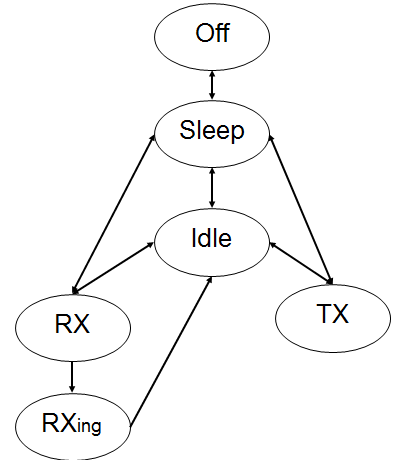


Figure 34. Radio state Machine.

The library name that can be called in the configuration XML file are:

* Transceiver\_radio\_half1d
* Transceiver\_radio\_half1d\_Wifi
* Transceiver\_radio\_802\_15\_4\_2400\_oqpsk
* Transceiver\_radio\_802\_15\_4\_868\_bpsk
* Transceiver\_radio\_802\_15\_4\_902\_bpsk
* Transceiver\_radio\_halfduplex\_spectrum

#### Methods Template

They are the methods that should be implemented by a transceiver model.

typedef struct \_transceiver\_methods {

void (\*rx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

void (\*tx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

int (\*set\_header) (call\_t \*to, call\_t \*from, packet\_t \*packet, destination\_t \*dst);

int (\*get\_header\_size) (call\_t \*to, call\_t \*from);

int (\*get\_header\_real\_size) (call\_t \*to, call\_t \*from);

void (\*tx\_end) (call\_t \*to, call\_t \*from, packet\_t \*packet);

void (\*cs) (call\_t \*to, call\_t \*from, packet\_t \*packet);

double (\*get\_noise) (call\_t \*to, call\_t \*from);

double (\*get\_cs) (call\_t \*to, call\_t \*from);

double (\*get\_power) (call\_t \*to, call\_t \*from);

void (\*set\_power) (call\_t \*to, call\_t \*from, double power);

int (\*get\_channel) (call\_t \*to, call\_t \*from);

void (\*set\_channel) (call\_t \*to, call\_t \*from, int channel);

classid\_t (\*get\_modulation) (call\_t \*to, call\_t \*from);

void (\*set\_modulation) (call\_t \*to, call\_t \*from, classid\_t modulation);

uint64\_t (\*get\_Tb) (call\_t \*to, call\_t \*from);

uint64\_t (\*get\_Ts) (call\_t \*to, call\_t \*from);

void (\*set\_Ts) (call\_t \*to, call\_t \*from, uint64\_t Ts);

double (\*get\_sensibility) (call\_t \*to, call\_t \*from);

void (\*set\_sensibility) (call\_t \*to, call\_t \*from, double sensibility);

void (\*sleep) (call\_t \*to, call\_t \*from);

void (\*wakeup) (call\_t \*to, call\_t \*from);

void (\*switch\_rx) (call\_t \*to, call\_t \*from);

void (\*switch\_idle) (call\_t \*to, call\_t \*from);

int (\*get\_modulation\_bit\_per\_symbol) (call\_t \*to, call\_t \*from);

int (\*get\_modulation\_type) (call\_t \*to, call\_t \*from);

void (\*set\_modulation\_type) (call\_t \*to, call\_t \*from, int modulation\_type);

} transceiver\_methods\_t;

#### Half1d

Library name: **transceiver\_radio\_half1d**  
  
Description:  This model is a simple half duplex radio interface i.e. a transceiver that cannot transmit and receive simultaneously. The capture effect is implemented: if the radio is locked on a signal but receives a stronger signal, it loses the first on and locks on the second one.

A packet can be transmitted only in TX mode and a packet can be received only in RXing mode. This model starts in SLEEP mode. Thus the transceiver must be wakeup by the MAC layer at the beginning of the simulation. After a transmission or a reception, the transceiver goes back in IDLE mode.

This model does not define a PHY layer header.

At the end of the simulation, the transceiver model prints the duration spent in the different mode and the energy consumption. This part could be exported in energy model.

Global Class parameter:

Local Node parameter:

**sensibility:**radio sensibility in dBm. Default value is "-92"dBm.

**T\_s:**time required to send a symbol. Default value is "4000"ns.

**channel:**radio channel. Default value is channel "0".

**dBm:**transmission power in dBm. Default value is "0"dBm.

**modulation:**modulation used by the radio. No default value, MUST be specified.

**current\_draw\_sleep:**Current in SLEEP mode (in mA). Default value is from CC2420 i.e. 20nA

**current\_draw\_idle:**Current in IDLE mode (in mA). Default value is from CC2420 i.e. 426µA

**current\_draw\_rx:**Current in RX/RXING mode (in mA). Default value is from CC2420 i.e. 18.8mA

**current\_draw\_tx:**Current in TX mode (in mA). Default value is from CC2420 i.e 17.4mA

#### Half1d\_Wifi

Library name: **transceiver\_radio\_half1d\_Wifi**  
  
Description:  This model is an IEEE 802.11 half duplex radio interface i.e. a transceiver that cannot transmit and receive simultaneously. The capture effect is implemented: if the radio is locked on a signal but receives a stronger signal, it loses the first on and locks on the second one.

A packet can be transmitted only in TX mode and a packet can be received only in RXing mode. This model starts in SLEEP mode. Thus the IEEE 802.11 transceiver must be wakeup by the MAC layer at the beginning of the simulation. After a transmission or a reception, the transceiver goes back in IDLE mode.

This model also adds automatically the PHY layer header:

* 10 Bytes for the sync
* 2 bytes for the SFD
* 4 bytes for the PLCP
* 4 bytes for the FCS

At the end of the simulation, the IEEE802.11 transceiver model prints the duration spent in the different mode and the energy consumption. This part could be partly exported in an energy model.

Global Class parameter:

Local Node parameter:

**sensibility:**radio sensibility in dBm. Default value is "-90"dBm.

**T\_s:**time required to send a symbol. Default value is "91"ns for 11 Mbps

**channel:**radio channel. Default value is channel "0".

**dBm:**transmission power in dBm. Default value is "0"dBm.

**modulation:**modulation used by the radio. No default value, MUST be specified.

**current\_draw\_sleep:**Current in SLEEP mode (in mA). Default value is from CC3000 i.e. 5µA

**current\_draw\_idle:**Current in IDLE mode (in mA). Default value is from CC3000 i.e. 5µA

**current\_draw\_rx:**Current in RX/RXING mode (in mA). Default value is from CC3000 i.e. 92mA

**current\_draw\_tx:**Current in TX mode (in mA). Default value is from CC3000 i.e 260mA

#### 802.15.4 (868MHz - BPSK, 902MHz - BPSK, 2.4GHz - OQPSK)

Library name: **transceiver\_radio\_802\_15\_4\_868\_bpsk/ transceiver\_radio\_802\_15\_4\_902\_bpsk/ transceiver\_radio\_802\_15\_4\_2400\_oqpsk**  
Description:  This model is an IEEE 802.15.4 compliant radio interface. The radio parameters are specified by the 802.15.4 standard. 3 configurations are available:

* **868MHz bpsk**
  + 868-868.6 MHz
  + BPSK
  + 300kchips/s, 20ksymbols/s, 20kb/s
  + sensibility -92dBm
* **902MHz bpsk** 
  + 902-928 MHz
  + BPSK
  + 600kchips/s, 40kb/s, 40ksymbols/s
  + sensibility -92dBm
* **2400MHz bpsk**
  + 2400-2483.5 MHz
  + O-QPSK
  + 2000kchips/s, 250kb/s, 62.5ksymbols/s
  + sensibility -85dBm

The IEEE802.15.4 transceiver is half duplex radio i.e. it can con receive and transmit in the same time. If the radio is not correctly configured, some packets can be lost. The capture effect is implemented in cs function : if the radio is locked on a signal but receives a stronger signal, it loses the first on and locks on the second one. The rx function is filtering on the PER.

A packet can be transmitted only in TX mode and a packet can be received only in RXing mode. This model starts in SLEEP mode. Thus the IEEE 802.15.4 transceiver must be wakeup by the MAC layer at the beginning of the simulation. After a transmission or a reception, the transceiver goes back in IDLE mode.

This model also adds automatically the PHY layer header:

* 4 Bytes for the preamble
* 1 byte for the SFD
* 1 byte for the frame length

At the end of the simulation, the IEEE802.15.4 transceiver model prints the duration spent in the different mode and the energy consumption. This part could be partly exported in an energy model.

Global Class parameter:

Local Node parameter:

**dBm:**transmission power in dBm. Default value is "-15"dBm.

**log\_status:** Flag to print log information: "0" i.e. no print and “1” for print. Default value is "0" i.e. no print.

**sensibility:**radio sensibility in dBm. Default value is "-92"dBm for BPSK\_868 and BPSK\_902 and "-85"dBm for OQPSK\_2400

**channel:**radio channel. Default value is channel "0".

**current\_draw\_sleep:**Current in SLEEP mode (in mA). Default value is from CC1100 i.e. 400nA for BPSK\_868 and BPSK\_902 and from CC2420 i.e. 20nA for OQPSK\_2400

**current\_draw\_idle:**Current in IDLE mode (in mA). Default value is from CC1100 i.e. 160µA for BPSK\_868 and BPSK\_902 and from CC2420 i.e. 426µA for OQPSK\_2400

**current\_draw\_rx:**Current in RX/RXING mode (in mA). Default value is from CC1100 i.e. 15.1mA for BPSK\_868 and BPSK\_902 and from CC2420 i.e. 18.8mA for OQPSK\_2400

**current\_draw\_tx:**Current in TX mode (in mA). Default value is from CC1100 i.e. 31mA for BPSK\_868 and BPSK\_902 and from CC2420 i.e. 9.9mA for OQPSK\_2400

**RSSI\_min\_value :** Minimum detectable received power for RSSI computation. Default value is 10.

**RSSI\_max\_value:** Maximal detectable received power for RSSI computation. Default value is 40.

**RSSI\_nbr\_bits**: Number of bits coding the value of RSSI. Default value is 8 bits.

**RSSI\_sigma:** Standard deviation of the RSSI value.Default value is 2.

Note that the modulation used by the radio is directly inserted in the compilation.

#### Transceiver Radio half duplex spectrum

Library name: **transceiver\_radio\_half\_duplex\_spectrum**

Description:  This model is an 802.15.4 compliant radio interface. Its characteristics are the following:

* The data rate is 50kb/s (for 100kHz bandwidth)
* The sensitivity is -110 dBm
* 1 channel in the 868 MHz band
* The energy consumption is
  + TX mode: 47 mA at 14 dBm
  + RX mode: 17 mA
  + IDLE mode: 1.3 mA
  + SLEEP mode: 0.5µA

This transceiver is half duplex radio i.e. it can con receive and transmit in the same time. If the radio is not correctly configured, some packets can be lost.

The transceiver module models the radio state machine and its different functioning modes:

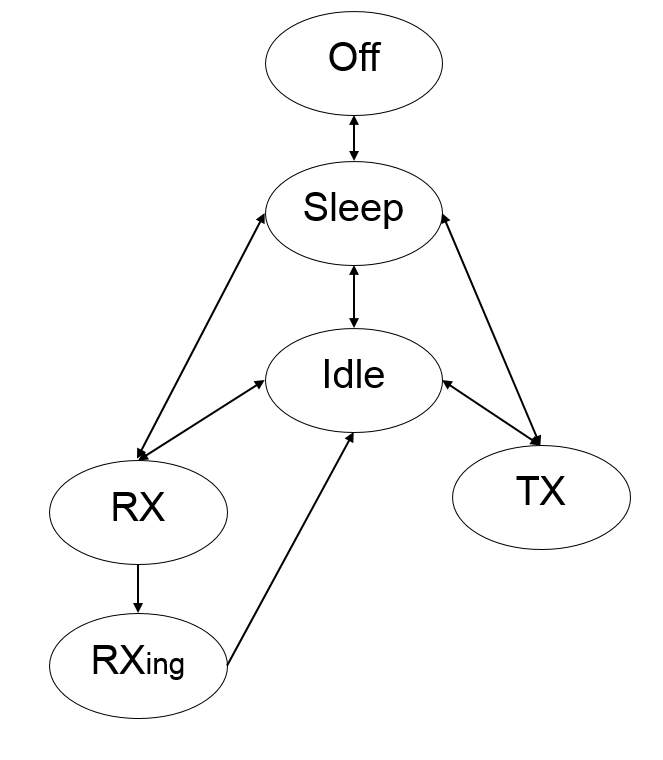


Figure 35: Possible radio states of the transceiver

A packet can be transmitted only in TX mode and a packet can be received only in RXing mode. This model starts in SLEEP mode. Thus the transceiver must be wakeup by the MAC layer at the before any possible reception or transmission. After a transmission or a reception, the transceiver goes back in IDLE mode.

At the end of the simulation, the transceiver model prints the duration spent in the different modes and the energy consumptions along with the ratio of each one of them.

This transceiver is partially capable of working with PHY models, i.e. it can use some of the functionalities of the 4.0 version. It is able to ask the PHY model to register and unregister for RX on the spectrum. For that purpose, it creates a fake packet with a primitive (already known by the PHY) as one of its fields dedicated to ask the PHY to register and unregister the node. Moreover, it no longer has capture effect functionalities (now SignalTracker is responsible for that). The channel sensing capability is also limited as there is no such functionality on PHY models until now.

Global Class parameter:

* **log\_status\_class:** Flag to print log information regarding all links: "0" i.e. no print and “1” for print. Default value is "0" i.e. no print.

Local Node parameter:

* **dBm:**transmission power in dBm. Default value is "14"dBm.
* **log\_status:** Flag to print log information at the end of the simulation regarding the time and energy spent on which state: "0" i.e. no print and “1” for print. Default value is "0" i.e. no print.
* **sensibility:**radio sensibility in dBm. Default value is "-110"dBm
* **current\_draw\_sleep:**Current in SLEEP mode (in mA). Default for CC1125 is 0.5µA
* **current\_draw\_idle:**Current in IDLE mode (in mA). Default for CC1125 is 1.3 mA
* **current\_draw\_rx:**Current in RX/RXING mode (in mA). Default for CC1125 is 17 mA
* **current\_draw\_tx:**Current in TX mode (in mA). Default for CC1125 is 47 mA (TX power of 14 dBm)

### Modulations

The library name that can be called in the configuration XML file are:

* modulation\_none
* modulation\_step
* modulation\_bpsk
* modulation\_classic\_oqpsk
* modulation\_oqpsk
* modulation\_mqam
* modulation\_fsk

#### Methods Template

They are the methods that should be implemented by a modulation model.

typedef struct \_modulation\_methods {

double (\* modulate) (call\_t \*to, call\_t \*from, double snr);

int (\* bit\_per\_symbol) (call\_t \*to, call\_t \*from);

int (\*get\_modulation\_type) (call\_t \*to, call\_t \*from);

void (\*set\_modulation\_type) (call\_t \*to, call\_t \*from, int modulation\_type);

} modulation\_methods\_t;

#### None

Library name: **modulation\_none**  
  
Description: This modulation model does not support interferences. The returned bit error rate is:

* 0: if the snr is equal to MAX\_SNR (no interference).
* 0.5: otherwise.

With the usage of noise model, the bit error rate will be automatically 0.5.

Global Class parameter:

Local Node parameter:

#### STEP

Library name: **modulation\_step**  
  
Description:  This model is a step modulation model. The returned bit error rate is:

* 0: if the snr is superior or equal to the step parameter.
* 0.5: otherwise.

Global Class parameter:

**step**: the snr threshold for the bit to be correctly received. Default value is 2.

Local Node parameter:

#### BPSK

Library name: **modulation\_bpsk**  
  
Description:  This model is the BPSK modulation model. The returned bit error rate is:

* 0: if the snr is equal to MAX\_SNR.
* Otherwise.

Global Class parameter:

Local Node parameter:

#### classic\_OQPSK

Library name: **modulation\_classic\_oqpsk**  
  
Description:  This model is the OQPSK modulation model. The returned bit error rate is:

* 0: if the snr is equal to MAX\_SNR.
* Otherwise.

Each symbol of the classic OQPSK modulation represents 2 bits.

Global Class parameter:

Local Node parameter:

#### OQPSK

Library name: **modulation\_oqpsk**  
  
Description:  This model is the IEEE 802.15.4 half-sine-shaped Offset Quadrature Phase Shift Keying (O-QPSK) modulation model. The returned bit error rate is:

* 0: if the snr is equal to MAX\_SNR.
* Otherwise.

The IEEE 802.15.4 OQPSK modulation model is different to the classic one. Each symbol represents 4 bits.

Global Class parameter:

Local Node parameter:

#### MQAM

Library name: **modulation\_mqam**  
  
Description:  This model is the MQAM modulation model. The returned bit error rate is:

* 0: if the snr is equal to MAX\_SNR.
* Otherwise.

Each symbol of the classic MQAM modulation represents bits i.e from 2 for m\_QAM=4 to 8 from m\_QAM=256

Global Class parameter:

* **m\_QAM**: order of the modulation. Default value is "4". Possible modulation types are: 4, 8, 16, 32, 64, 128 and 256.

Local Node parameter:

#### FSK

Library name: **modulation\_fsk**  
  
Description:  This model is the FSK modulation model. The returned bit error rate is:

* 0: if the snr is equal to MAX\_SNR.
* Otherwise.

Each symbol of the FSK modulation represents 1 bit.

Global Class parameter:

Local Node parameter:

### Interface

The library name that can be called in the configuration XML file are:

* interface\_antenna\_ominidrectionnal

#### Methods Template

They are the methods that should be implemented by an interface model.

typedef struct \_interface\_methods {

void (\*rx) (call\_t \*to, call\_t \*from, packet\_t \*packet);

void (\*cs) (call\_t \*to, call\_t \*from, packet\_t \*packet);

double (\*get\_loss) (call\_t \*to, call\_t \*from);

angle\_t \* (\*get\_angle) (call\_t \*to, call\_t \*from);

void (\*set\_angle) (call\_t \*to, call\_t \*from, angle\_t \*angle);

double (\*gain\_tx) (call\_t \*to, call\_t \*from, position\_t \*pos);

double (\*gain\_rx) (call\_t \*to, call\_t \*from, position\_t \*pos);

mediumid\_t (\*get\_medium)(call\_t \*to, call\_t \*from);

int (\*get\_type) (call\_t \*to);

} interface\_methods\_t;

#### Omnidirectional

Library name: **interface\_antenna\_omnidirectionnal**  
  
Description:  This interface model is for an omnidirectional antenna.

Global Class parameter:

Local Node parameter:

* **loss:**loss induced to the signal by the RF antenna. Ex: the loss in a coaxial cable of length 0.1m at 2.45GHz is -0.09dB. Default value is "0".
* **gain-tx:** the antenna gain in emission (dB). Default value is "0".
* **gain-rx:**the antenna gain in reception (dB). Default value is "0".
* **angle-xy:**the antenna orientation in the xy plane (Radian between 0 and 2). Default value is "0".
* **angle-z:**the antenna orientation regarding the xy plane(Radian between 0 and 2). Default value is "0".
* **medium:** the medium to which the antenna is interfacing. This field is mandatory. If the medium parameter is not set, it returns an error.

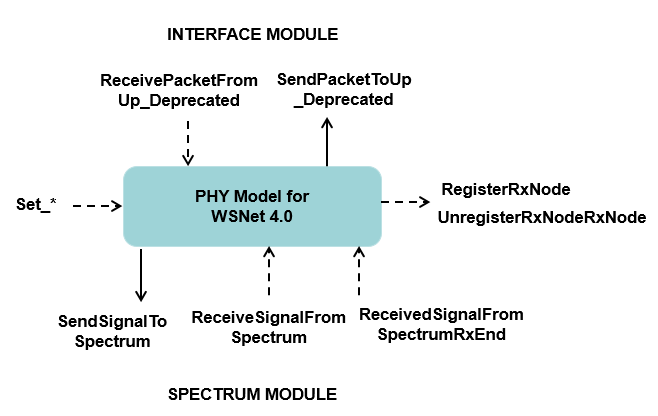
### PHY

In order to interface between the packet oriented higher layers and the new medium, a PHY model has been added.

The PHY model works as a sort of orchestrator of the physical layer events and models and also as an interface with the spectrum.

Each node willing to connect to the spectrum must have a PHY model. The relation with spectrums are: Spectrum has 1..N PhyModels connected to it whereas PhyModel has only 1 Spectrum on which it is connected.

Moreover, a node can have several PHY models, for instance, imagine a node has two transceivers one operating on WiFi and the other on LoRa, this means that we will need a PHY for the physical layer activity of both WiFi and LoRa. Rule of thumb is: for each transceiver a PHY is needed.



In the beginning of the simulation, the PHY model identify all the models under its umbrella, i.e. the Interference, Signal Tracker, Coding and Error models that are connected to it. After that, the PHY is able to call any of them whenever it finds necessary. In order to do so, in your bootstrap function you should add:

// set all the models connected to the phy

my\_object->SetSignalTrackerModel(TO\_CPP(get\_first\_object\_binding\_down\_by\_type(to,MODELTYPE\_SIGNAL\_TRACKER),SignalTrackerModel));

my\_object->SetErrorModel(TO\_CPP(get\_first\_object\_binding\_down\_by\_type(to,MODELTYPE\_ERROR),ErrorModel));

my\_object->SetInterferenceModel(TO\_CPP(get\_first\_object\_binding\_down\_by\_type(to,MODELTYPE\_INTERFERENCE),InterferenceModel));

my\_object->SetCodingModel(TO\_CPP(get\_first\_object\_binding\_down\_by\_type(to,MODELTYPE\_CODING),CodingModel));

And your xml file should correctly connect the PHY along with the models under its umbrella:

<phy name="phy">

<up name="antenna" />

<down name="signal\_tracker" />

<down name="interference" />

<down name="error" />

<down name="coding" />

</phy>

<error name="error">

<up name="phy" />

</error>

<coding name="coding">

<up name="phy" />

</coding>

<interference name="interference">

<up name="phy" />

</interference>

<signal\_tracker name="signal\_tracker">

<up name="phy" />

</signal\_tracker>

A list of recommendations for a developer when creating a new Phy model is described below. Notice that it is not mandatory, it is just a **recommendation** that can help you avoid common pitfalls:

* When you receive a packet from upper layer, remember to update the fields of packets with information regarding transmittion power, start and end time, etc . Notice that there are still several features that are still working with 3.1 version, so, for backward compatibility you should keep them there. One example is shown below:

packet->node = node->id;  
 packet->interface = from\_interface->classid;

packet->txdBm = transceiver\_get\_power(&from\_transceiver, from\_interface);

packet->channel = transceiver\_get\_channel(&from\_transceiver, from\_interface);

packet->modulation = transceiver\_get\_modulation(&from\_transceiver, from\_interface);

packet->Tb = transceiver\_get\_Tb(&from\_transceiver, from\_interface);

packet->duration = packet->real\_size \* packet->Tb;

packet->clock0 = get\_time();

packet->clock1 = packet->clock0 + packet->duration;

packet->rxdBm = 0;

packet->rxmW = 0;

packet->PER = 0;

packet->RSSI = 0;

packet->LQI = 0;

packet->SINR = 0;

* After that, create first the waveform with all the frequency intervals before creating the signal.
* When receiving a signal from the spectrum (RX\_BEGIN) add it to your map of received signals and don’t forget to verify if you can cast the signal to the type your PHY model is able to handle

auto rf\_signal = std::dynamic\_pointer\_cast<RFSignal>(signal);

if (rf\_signal) {

// it is a rf\_signal

} else {

// not a rf\_signal  
}

* After, update pathloss, shadowing and fading of the signal with the old style of the 3.1 version (This guarantees that both packet and signal will have all updated)

medium\_compute\_rxdBm(rf\_signal->GetPacket\_Deprecated(), &to, &from);

* Update the signal by converting the value of rxdBm into values of mW/Hz
* Verify with your signal tracker if it will take this signal or not. If the signal\_tracker chose this signal, we start the interface\_cs (as before on WSNET3.1)
* When receiving a signal from the spectrum (RX\_END), first check with the Signal Tracker if it is the selected signal, if this is the case, update the noise of your signal and the SNR value. (Set the SINR value to the same value, this is to prevent from the case the interference model does not update the SINR at all).
* After that, ask interference model to apply all interferences.
* The interference model will (probably) update the SINR value, use this value to calculate the BER and the PER for your packet and send it up.

The library name that can be called in the configuration XML file are:

* phy\_rf\_signal\_adjacent\_band

#### Methods Template

The class below is the abstract class of a PHY model, this means that we have some methods that need to be implemented if you ever want to inherit from this class. They are the methods that should be implemented by a PHY model. The public methods are the ones with public visibility and can be called by the kernel, other models, etc, whoever has access to the model.

/\*\* \brief The Abstract Base Class : PhyModel Class

This means that no instance of the PhyModel class can exist. Only classes which inherit from the PhyModel class can exist. This model is used to connect nodes to the spectrum. The relation with spectrums are: Spectrum has 1..N PhyModels connected to it whereas PhyModel has only 1 Spectrum on which it is connected.

\* \fn ReceiveSignalFromSpectrum is used to receive a signal from the spectrum

\* \fn SendSignalToSpectrum is used to send a signal to the spectrum

\* \fn RegisterRxNode is used to register a node on the Rx list of the spectrum

\* \fn UnregisterRxNode is used to remove a node of the Rx list of the spectrum

\* \fn ReceiveSignalFromSpectrumImpl implements the reception of a signal from the spectrum

\* \fn SendSignalToSpectrum implements the transmission of a signal to the spectrum

\* \fn RegisterRxNode implements how to register the node on the Rx list of the spectrum

\* \fn UnregisterRxNode implements how to remove the node of the Rx list of the spectrum

\* \fn ReceivedSignalFromSpectrumRxEnd is to be called on Rx End of the signal

\*\*/

class PhyModel : public std::enable\_shared\_from\_this<PhyModel>{

**public**:

PhyModel();

virtual ~PhyModel();

void ReceiveSignalFromSpectrum(std::shared\_ptr<Signal> signal);

void ReceivePacketFromUp\_Deprecated(call\_t \*to, call\_t \*from,packet\_t \*packet);

void SendPacketToUp\_Deprecated(std::shared\_ptr<Signal> signal);

void SendSignalToSpectrum(std::shared\_ptr<Signal> signal);

void RegisterRxNode(SetOfFrequencyIntervals freq\_intervals);

void UnregisterRxNode();

void ReceivedSignalFromSpectrumRxEnd(std::shared\_ptr<Signal> signal);

void SetInterferenceModel(InterferenceModel \*);

void SetModulatorModel(ModulatorModel \*);

void SetErrorModel(ErrorModel \*);

void SetCodingModel(CodingModel \*);

void SetSignalTrackerModel(SignalTrackerModel \*);

void SetSpectrum(SpectrumModel \*);

void Print() const;

PhyModellUid GetUID() const;

**private**:

virtual void **ReceiveSignalFromSpectrumImpl**(std::shared\_ptr<Signal>)= 0;

virtual void **ReceivePacketFromUp\_DeprecatedImpl**(call\_t \*to, call\_t \*from,packet\_t \*) = 0;

virtual void **SendPacketToUp\_DeprecatedImpl**(std::shared\_ptr<Signal>) = 0;

virtual void **SendSignalToSpectrumImpl**(std::shared\_ptr<Signal>)= 0;

virtual void **RegisterRxNodeImpl**(SetOfFrequencyIntervals) = 0;

virtual void **UnregisterRxNodeRxNodeImpl**() = 0;

virtual void **ReceivedSignalFromSpectrumRxEndImpl**(std::shared\_ptr<Signal> )=0;

virtual void **PrintImpl**() const;

static PhyModellUid uid\_counter\_;

PhyModellUid uid\_;

**protected**:

InterferenceModel \* interference\_model\_;

ModulatorModel \* modulator\_model\_;

ErrorModel \* error\_model\_;

CodingModel \* coding\_model\_;

SignalTrackerModel \* signal\_tracker\_model\_; // signal tracker is here temporarily. It should be on radio. Radio does not receive signals yet.

MapOfSignals received\_signals\_; // a map containing all received signal being treated at the moment

SpectrumModel \* spectrum\_; // The spectrum on which it is connected

std::shared\_ptr<RegisteredRxNode> registered\_rx\_node\_;

};

#### RF Signal adjacent band

Library name:  **phy\_rf\_signal\_adjacent\_band**  
  
Model class:

class RFSignalAdjacentBandPhyModel : public PhyModel{

public:

RFSignalAdjacentBandPhyModel(Frequency freq\_center, double factor\_adjacent, uint number\_adjacent\_bands,

Frequency delta\_adjacent\_bands, Frequency bandwidth, int log\_status, nodeid\_t nodeid) ;

~RFSignalAdjacentBandPhyModel();

SetOfFrequencyIntervals GetOperatingFrequencyIntervalsTx();

void UpdateFrequencyIntervalsTx (SetOfFrequencyIntervals freq\_intervals);

void UpdateFrequencyIntervalsRx (SetOfFrequencyIntervals freq\_intervals);

int GetLogStatus();

private:

void ReceiveSignalFromSpectrumImpl(std::shared\_ptr<Signal>);

void SendPacketToUp\_DeprecatedImpl(std::shared\_ptr<Signal>);

void SendSignalToSpectrumImpl(std::shared\_ptr<Signal>);

void RegisterRxNodeImpl(SetOfFrequencyIntervals);

void UnregisterRxNodeRxNodeImpl();

void ReceivedSignalFromSpectrumRxEndImpl(std::shared\_ptr<Signal> );

void PrintImpl() const;

void ReceivePacketFromUp\_DeprecatedImpl(call\_t \*to, call\_t \*from,packet\_t \* packet);

SetOfFrequencyIntervals **frequency\_intervals\_tx\_**; // the set of frequency intervals the phy model is operating

SetOfFrequencyIntervals **frequency\_intervals\_rx\_**; // the set of frequency intervals the phy model is operating

**Frequency freq\_center\_** = 868075000;

double **factor\_adjacent\_**;

uint **number\_adjacent\_bands\_** = 0;

Frequency **delta\_adjacent\_bands\_** = 1000;

Frequency **bandwidth\_** = 100000;

std::multimap<Time, SignallUid> **signals\_already\_treated\_**;

Time **evaluation\_window\_** = 0;

int **log\_status\_;**

nodeid\_t nodeid\_;

};

Description:  The PHY model works as a sort of orchestrator of the physical layer events and models and also as an interface with the spectrum.

The RFSignal adjacent band PHY model works with RFSignals, i.e. they are only able to receive RFSignals (and all Signals inheriting from it, e.g. LoraSignal).

The model is able to create signals with several adjacent bands that simulate a real-world signal that end up generating noise on the adjacent bands of which it was transmitted. The user of this model is able to choose the number of adjacent bands a signal will have whenever it is being transmitted. The user can also select the size of the frequency intervals (delta\_freq\_adjacent) of the adjacent bands. Finally, the user can also select the factor (reduction) that will be applied to the transmitted power to get the PSD of the frequency interval of the adjacent band.

The PSD value of both the central band and the adjacent bands are calculated respecting the following:

with **Ptx** the transmitted power, **f** the factor adjacent, **n** the number of adjacent bands, **BW** the bandwith of the central band, **ΔF** the frequency interval of the adjacent bands, and **PSD** the power spectral density of the interval.

An illustrative example with 2 adjacent bands is given below.

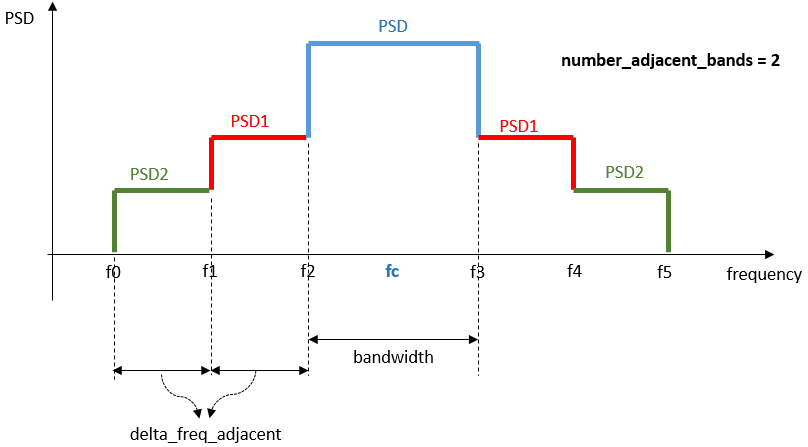


Figure 36: Example of generated signal with its adajacent bands

The recommendation is that whenever the simulation does not involve multi-band signals being transmitted at the same time, users should set the number\_adjacent\_bands to 0, as there will be no other frequencies being used, which makes useless the use of adjacent bands. In this way, the simulations will consume less memory as well as less time, as there is no need to keep the extra adjacent bands and there is no need to perform interval searches on them.

Moreover, the model uses an evaluation window to avoid storing all signals received until the end of the simulation. The idea is to have a window that will evolve, once the signal is no longer inside this window, we can remove it from our set, as it will no longer cause any interference in others. This helps reducing overhead calculations for the interference model and also reduces the memory requirements of the program.

The evaluation window moves and increases or decreases its size according to the signals being received. Once it is verified that the signal will no longer interfere with others (there is no intersection), the signal is removed from the set of received signals and the evaluation window start is updated the figures below show an example of the functioning of the evaluation window. We start with 3 signals inside the evaluation window.

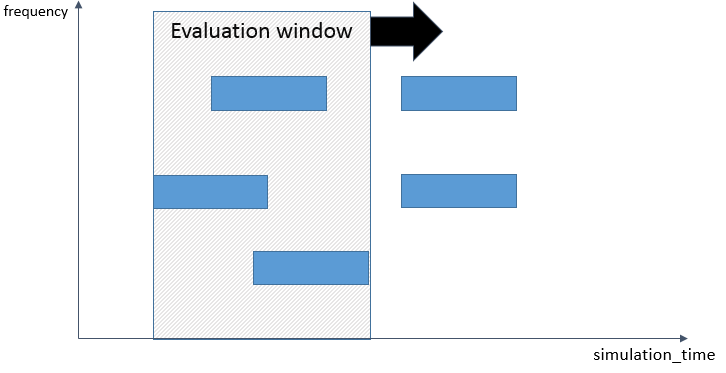


Figure 37: Example of the evaluation window moving in time

After the signals were treated and verified that they are no longer able to interfere with othe signals, they are removed from the set of received signals and the evaluation window moves in time, as veritied below.

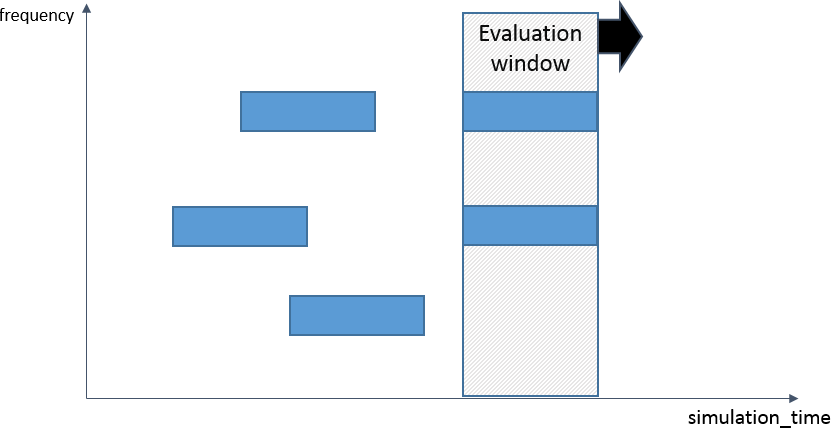


Figure 38: Example of the evaluation window moving in time

Global Class parameter:

Local Node parameter:

* **bandwidth:** the bandwidth on which it will operate. Default value is “100000”
* **freq\_center:** the frequency center on which it will operate. Default value is “868075000”.
* **number\_adjacent\_bands:** the bandwidth on which it will operate. Default value is “0”
* **delta\_freq\_adjacent:** the bandwidth of the adjacent bands. Default value is “1000”
* **factor\_adjacent:** the attenuation factor of the adjacent bands. Default value is “0.0”

### Spectrum

The Spectrum model aims at providing support for modeling both time and frequency-dependent aspects of communications. The spectrum model basically works as the medium where signals are transmitted and received. Spectrum models are responsible for scheduling RX\_BEGIN, RX\_END and TX\_END events on the scheduler.

They can create algorithms that will properly filter the nodes which will receive a given signal according to their needs. They can be created to work with specific types of signals, or with specific bands, etc.

A list of recommendations for a developer when creating a new Spectrum model is described below. Notice that it is not mandatory, it is just a recommendationthat can help you avoid common pitfalls:

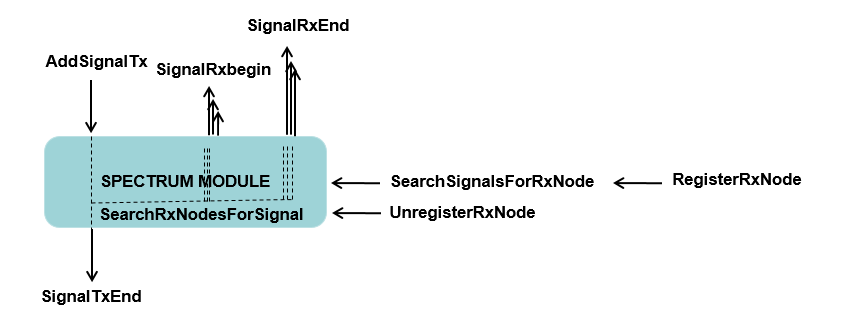
* Remember to keep track of which nodes are registered to RX and which signals are being transmitted. You can use an IntervalSearchTree to help you with the task of finding overlaps.
* During a RegisterRXNode call, remember to verify whether there are some signals already being transmitted (Use the IntervalSearchTree to find signals for this node)
* During a transmittion event remember to find the registered nodes (using the IntervalSearchTree) that overlap with this signal. After that, make a copy of the signal to each one of the nodes eligible to receive the signal and create the RX\_BEGIN and RX\_END events.
* Don’t forget to remove unregistered nodes and also signals already finished.

The library name that can be called in the configuration XML file are:

* spectrum\_multiband\_rf

#### Methods Template

They are the methods that should be implemented by a spectrum model.



/\*\* \brief The Abstract Base Class : SpectrumModel Class

This means that no instance of the SpectrumModel class can exist. Only classes which inherit from the SpectrumModel class can exist

The relation with PHY models are: Spectrum has 1..N PhyModels connected to it whereas PhyModel has only 1 Spectrum on which it is connected.

\* \fn UnregisterRXNode is used to unregister a node on the spectrum

\* \fn AddSignalTx is used to notify the spectrum a signal will be transmitted

\* \fn SignalRxBegin is called by the scheduler when a given signal will start to be received

\* \fn SignalRxEnd is called by the scheduler when the reception of a given signal is at the end

\* \fn SignalTxEnd is called by the scheduler when the transmittion of a given signal is at the end

\*\*/

class SpectrumModel{

**public**:

SpectrumModel();

virtual ~SpectrumModel();

std::vector<std::shared\_ptr<Signal>> RegisterRXNode(std::weak\_ptr<RegisteredRxNode> rx\_node);

void UnregisterRXNode(std::weak\_ptr<RegisteredRxNode> rx\_node);

void AddSignalTx(std::shared\_ptr<Signal> signal);

void SignalRxBegin(std::shared\_ptr<Signal> signal);

void SignalTxEnd(std::weak\_ptr<Signal> signal);

void SignalRxEnd(std::shared\_ptr<Signal> signal);

SpectrumUid GetUID() const;

**private**:

//virtual RxNodeFilter **GetRegisteredNodeByIdImpl**(nodeid\_t) = 0;

virtual std::vector<std::shared\_ptr<Signal>> **RegisterRxNodeImpl**(std::weak\_ptr<RegisteredRxNode>)=0;

virtual void **UnregisterRxNodeImpl**(std::weak\_ptr<RegisteredRxNode>)=0;

virtual void **SignalAddTxImpl**(std::shared\_ptr<Signal>) = 0;

virtual void **SignalRxBeginImpl**(std::shared\_ptr<Signal>) = 0;

virtual void **SignalTxEndImpl**(std::weak\_ptr<Signal> signal) = 0;

virtual void **SearchRxNodesForSignalImpl**(std::weak\_ptr<Signal>) = 0;

virtual std::vector<std::shared\_ptr<Signal>> **SearchSignalsForRxNodeImpl**(std::weak\_ptr<RegisteredRxNode>) = 0;

virtual void **SignalRxEndImpl**(std::shared\_ptr<Signal>) = 0;

**protected:**

void SearchRxNodesForSignal(std::weak\_ptr<Signal> signal);

std::vector<std::shared\_ptr<Signal>> SearchSignalsForRxNode(std::weak\_ptr<RegisteredRxNode> rx\_node);

static SpectrumUid uid\_counter\_;

SpectrumUid uid\_;

};

#### Multi-band RF

Library name: **spectrum\_multiband\_rf**

Model class:

class MultiBandRFSpectrumModel : public **SpectrumModel**{

public:

MultiBandRFSpectrumModel(std::unique\_ptr<IntervalTree>,

std::unique\_ptr<IntervalTree>,

std::unique\_ptr<RangeTree> ,

RegisterMode);

MultiBandRFSpectrumModel();

~MultiBandRFSpectrumModel(){};

private:

std::vector<std::shared\_ptr<Signal>> RegisterRxNodeImpl(std::weak\_ptr<RegisteredRxNode>);

void UnregisterRxNodeImpl(std::weak\_ptr<RegisteredRxNode>);

void SignalAddTxImpl(std::shared\_ptr<Signal>);

void SignalRxBeginImpl(std::shared\_ptr<Signal>);

void SignalTxEndImpl(std::weak\_ptr<Signal> signal);

void SearchRxNodesForSignalImpl(std::weak\_ptr<Signal>);

std::vector<std::shared\_ptr<Signal>> SearchSignalsForRxNodeImpl(std::weak\_ptr<RegisteredRxNode>);

void SignalRxEndImpl(std::shared\_ptr<Signal>);

std::map<nodeid\_t, std::weak\_ptr<RegisteredRxNode>> **rx\_nodes\_registered\_**;

std::map<SignallUid, std::weak\_ptr<RFSignal>> **txing\_signals\_**;

std::unique\_ptr<IntervalTree> **rx\_nodes\_search\_tree\_**;

std::unique\_ptr<IntervalTree> **txing\_signals\_search\_tree\_**;

std::unique\_ptr<RangeTree> **range\_tree\_;**

RegisterMode **register\_mode\_**;

};

Description:  This spectrum model implements a multi-band spectrum where RFSignals are transmitted and received. It basically saves all registered nodes in a map and all of the intervals they are listening in an IntervalSearchTree (rx\_nodes\_search\_tree\_). The model also save all signals being transmitted in a map as well as all intervals that are being transmitted in an IntervalSearchTree (txing\_signals\_search\_tree\_). Thus, whenever a new RFSignal is transmitted, the spectrum model first creates a TX\_END event and then it saves the signal on the map and all its intervals on the txing\_signals\_search\_tree\_. After that, it checks the rx\_nodes\_search\_tree\_for all intervals that are overlapping with the current signal being transmitted. In this way, it is able to identify all registered nodes that are listening to a band on which this signal is being transmitted. Then, with all the filtered registered nodes that were listening at that time and it creates copies of the signal to be sent to each PHY model of the filtered nodes. Subsequently, it creates the RX\_BEGIN event of the scheduler as well as the RX\_END event. When the RX\_BEGIN event takes place, it forward the copied signal to the appropriate PHY model ReceiveSignalFromSpectrum method. When the RX\_END event takes place, it calls the appropriate PHY model ReceivedSignalFromSpectrumRxEnd method.

At the TX\_END event, it will delete the signal from the map of transmitting signal and it will also delete all intervals belonging to the interval on the txing\_signals\_search\_tree\_.

Another important scenario is when a node is registering. During this process, the spectrum model firstly updates the map of registered nodes and rx\_nodes\_search\_tree\_. Secondly it checks whether there is already signals which has overlapping intervals with the registered intervals. It then returns to the calling PHY model all signals that were found on the given band. Notice that in this case the spectrum model **DOES NOT create RX\_BEGIN** events as they already happened at this time, it only creates RX\_END events. PHY models willing to work with this spectrum model should take this into consideration.

Global Class parameter:

Local Node parameter:

### Signal tracker

The signal tracker model is responsible for verifying which signal is going to be the signal of interest, or the tracked signal. This model is under the umbrella of a PHY models, meaning that it will be called upon request of the PHY model. The PHY model, when necessary, will call the ReceiveSignal() method passing a given received signal. The job of the signal tracker model is to verify whether the signal will be the one that the transceiver will decode, or it will be considered as an interferer for another signal already in track.

Later, usually when an event of RX\_END arrives, the PHY can ask whether a given signal was the one selected by calling the VerifySignalIsSelected() , which the signal tracker models will return as true or false. The PHY can also request the signal tracker model the current selected signal by using the GetSelectedSignal() method (maybe deprecated). Finally, the ResetSelectedSignal() is used to clear the current select signal, this is mostly used when the node is no longer listening or putting its radio into sleep mode, so that the RX\_END will no longer be valid.

The library name that can be called in the configuration XML file are:

#### Methods Template

They are the methods that should be implemented by a signal tracker model.

/\*\* \brief The Abstract Base Class : SignalTrackerModel Class

This means that no instance of the Interval class can exist. Only classes which inherit from the Interval class can exist.

\* \fn GetSelectedSignal() return the selected signal

\* \fn ResetSelectedSignal() resets the selected signal

\* \fn VerifySignalIsSelected() checks if the signal is selected or not

\* \fn ReceiveSignal() receive a new signal

\* \fn Print() prints information about the SignalTrackerModel

\* \fn GetUID() return the UID of the SignalTrackerModel

\* \fn VerifySignalIsSelectedImpl() implements the method that checks if the signal is selected or not (to be implemented by derived class)

\* \fn GetSelectedSignalImpl() implements the function that get the selected the signal that will be received (to be implemented by derived class)

\* \fn ResetSelectedSignalImpl() implements the function that resets the selected the signal (to be implemented by derived class)

\* \fn ReceiveSignalImpl() implements the function that selects the signal that will be received (to be implemented by derived class)

\* \fn PrintImpl() implemetns the Print function (to be implemented by derived class)

\*\*/

class SignalTrackerModel {

**public**:

SignalTrackerModel();

virtual ~SignalTrackerModel();

std::shared\_ptr<Signal> GetSelectedSignal();

bool VerifySignalIsSelected(std::shared\_ptr<Signal> signal);

bool ReceiveSignal(std::shared\_ptr<Signal> signal);

void ResetSelectedSignal();

void Print() const;

SignalTrackerModelUid GetUID() const;

**private**:

virtual std::shared\_ptr<Signal> **GetSelectedSignalImpl**() =0 ;

virtual void **ResetSelectedSignalImpl**() = 0;

virtual void **PrintImpl**() const;

virtual bool **ReceiveSignalImpl**(std::shared\_ptr<Signal>) = 0;

virtual bool **VerifySignalIsSelectedImpl**(std::shared\_ptr<Signal>) = 0;

static SignalTrackerModelUid uid\_counter\_;

SignalTrackerModelUid uid\_;

};

#### Pick first RF

Library name: **signal\_tracker\_pick\_first\_rf**

Model class:

class PickFirstSignalTrackerModel : public SignalTrackerModel {

public:

PickFirstSignalTrackerModel();

PickFirstSignalTrackerModel(Frequency);

~PickFirstSignalTrackerModel();

private:

std::shared\_ptr<Signal> GetSelectedSignalImpl();

bool ReceiveSignalImpl(std::shared\_ptr<Signal>);

bool VerifySignalIsSelectedImpl(std::shared\_ptr<Signal>);

void ResetSelectedSignalImpl();

void PrintImpl() const;

Frequency **freq\_center**\_;

std::weak\_ptr<Signal> **current\_signal\_;**

};

Description:  Select the first arrived RFSignal and stick to it until the end of the transmission regardless of the link quality or SNR. This simple example does not check bandwidth nor the frequency center of the RFSignal.

Global Class parameter:

Local Node parameter:

### Pathloss

#### Method Template

The library name that can be called in the configuration XML file are:

* pathloss\_none
* pathloss\_freespace
* pathloss\_twowayground
* pathloss\_range
* pathloss\_pathloss\_wiplan
* pathloss\_itu\_indoor
* pathloss\_filestatic
* pathloss\_mixed

They are the methods that should be implemented by a pathloss model.

typedef struct \_pathloss\_methods {

double (\*pathloss) (call\_t \*to\_pathloss, call\_t \*to\_interface, call\_t \*from\_interface, packet\_t \*packet, double rxdBm);

} pathloss\_methods\_t;

#### None

Library name: **pathloss\_none**  
  
Description:  This pathloss model does not apply a pathloss. The returned received power is the same.

Global Class parameter:

Local Node parameter:

#### Free-space

Library name: **pathloss\_freespace**  
  
Description:  This model simulates a path loss in free-space depending on the distance between nodes.

In the *freespace propagation model*, the average power received at distanceis computed as follows:



Where is the transmission power (e.g. 0dBm) and is the pathloss experienced at the actual distance, with the pathloss decay exponent and the pathloss at a reference distance(e.g. 1m), the center frequency and the speed of light.

Global Class parameter:

* **frequency\_MHz**: frequency used in the Friis formula in MHz. Default value is "868"MHz.
* **pathloss :** pathloss exponent used in the Friis formula. Default value is "2".
* **Pr0 :** pathloss at reference distance (d0=1 meter) used in the Friis formula. Default value is

Local Node parameter:

#### 2-Ray Ground

Library name: **pathloss\_tworayground**  
  
Description:  This propagation model is a two-Ray ground propagation model. This model assumes a perfect flat ground. In the *two-ray ground reflection* propagation model, the average power received at distanceis computed as follows:



Where and are the heights of the TX and RX antennas, respectively, and is the cross-over distance which is calculated as: .

Global Class parameter:

* **frequency\_MHz**: frequency used in the Friis formula in MHz. Default value is "868"MHz.

Local Node parameter:

#### ITU indoor model

Library name: **pathloss\_itu\_indoor**  
  
Description:  This propagation model is an ITU indoor propagation model. The returned received power is:

* rxdBm – L\_dB

Where

Global Class parameter:

* **frequency\_MHz**: the carrier frequency in MHz. Default value is "868"MHz.
* **Lf:**the floor penetration loss factor. Default value is "-1" and will be set to 15+4\*(n\_floors-1)
* **n\_floors:**the number of floors penetrated. Default value is "2".
* **distpower\_lc:**the distance power loss coefficient. Default value is "33"m.

Local Node parameter:

#### Pathloss\_wiplan

Library name: **pathloss\_pathloss\_wiplan**  
  
Description:  This propagation model read power from the wiplan out file. The returned received power is the one in the wiplan file.

Global Class parameter:

* **file:**the file giving the received power of each link. Default value is "wiplanout.xml".

Local Node parameter:

#### Range

Library name: **pathloss\_range**  
  
Description:  This propagation model is a disk model. No attenuation of the signal for a distance up to range. At a further distance, the signal is lost.

* rxdBm if distance is less than range
* MIN\_DBM otherwise

Global Class parameter:

* **range:**range of the signal. Default value is "10" meters.

Local Node parameter:

#### Filestatic.

Library name: **pathloss\_filestatic**  
  
Description:  This propagation model takes as input a file giving the link probability success for each link. Whenever the propagation model is called, a random value is drawn to see whether the information successfully crossed the link. The returned received power is:

* rxdBm if random is less than the link probability success, ie. no attenuation of the signal
* MIN\_DBM otherwise i.e. maximal attenuation.

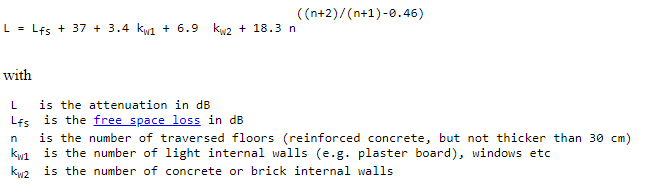
Global Class parameter:

* **file:**the file giving the link success probability. Default value is "propagation.data".

Local Node parameter:

#### COST231

Library name: **pathloss\_cost231**  
  
Description:  This propagation model is based on the COST231 project. The pathloss attenuation L is defined as follows:



The calculation of the number of walls separating two nodes is performed by models map and link.

Global Class parameter:

* **pathloss\_exponent\_los:**pathloss exponent used in the Friis formula for the free space loss. Default value is "2".
* **pathloss\_PL0\_los:** The pathloss at the reference distance for the LOS radio link condition (in dBm) used in the Friis formula for the free space loss. Default value is "0".
* **pathloss\_dist0\_los:** The reference distance for the pathloss model for the LOS radio link condition (in meters) used in the Friis formula for the free space loss. Default value is "1.0".
* **pathloss\_frequency:** pathloss frequency in MHz (the main/center frequency of the signal) used in the Friis formula for the free space loss. Default value is "868".
* **wall\_type:** The wall type ( Use value “0” for kw1 or value “1” for kw2. Default value is "1".

Local Node parameter:

#### **Mixed Pathloss**

Library name: **pathloss\_mixed**  
  
Description:

The mixed pathloss simulation module provides two main pathloss propagation models with parameters depending on the link condition: the *freespace* model and the *two-ray ground reflection* model.

In the *freespace propagation model*, the average power received at distanceis computed as follows:



Where is the transmission power (e.g. 0dBm), and are the TX and RX antenna Gains (e.g. 0dBi), respectively, and is the pathloss experienced at the actual distance , with the pathloss decay exponent and the pathloss at a reference distance(e.g. 1m), the center frequency (e.g. 2.4GHz) and the speed of light.

The parameter depending on the link condition are the pathloss at the reference distance, the reference distance and the pathloss decay exponent of the pathloss propagation model. For example, may be equal to 2, 3 and 3.3 in case of LOS, NLOS and NLOS2 radio link conditions, respectively

In the *two-ray ground reflection* propagation model, the average power received at distanceis computed as follows:



Where and are the heights of the TX and RX antennas, respectively, and is the cross-over distance which is calculated as: .

Global Class parameter:

* **pathloss:** this defines the pathloss propagation model to be considered. Possible values are: “freespace”, “tworayground” or “none”. By default, the pathloss propagation model is freespace.
* **Pathloss\_frequency:** this defines the frequency band for the pathloss. Default value is 2400 MHz
* **pathloss\_exponent\_los:** this defines the pathloss decay exponent in case of LOS radio link condition. By default, this parameter is equal to 2.
* **pathloss\_exponent\_nlos:** this defines the pathloss decay exponent in case of NLOS radio link condition. By default, this parameter is equal to 3.3.
* **pathloss\_exponent\_nlos2:** this defines the pathloss decay exponent in case of NLOS2 radio link condition. By default, this parameter is equal to 3.3.
* **pathloss\_PL0\_los:** this defines the pathloss at the reference distance in case of LOS radio link condition. By default, this parameter is equal to 0.
* **pathloss\_PL0\_nlos:** this defines the pathloss at the reference distance in case of NLOS radio link condition. By default, this parameter is equal to 0.
* **pathloss\_PL0\_nlos2:** this defines the pathloss at the reference distance in case of NLOS2 radio link condition. By default, this parameter is equal to 0.
* **pathloss\_dist0:** this defines the reference distance for the freespace propagation model. By default, this parameter is equal to 1 (m).
* **pathloss\_ dist0\_los:** this defines defines the reference distance in case of LOS radio link condition. By default, this parameter is equal to 0.
* **pathloss\_ dist0\_nlos:** this defines the reference distance in case of NLOS radio link condition. By default, this parameter is equal to 0.
* **pathloss\_ dist0\_nlos2:** this defines the reference distance in case of NLOS2 radio link condition. By default, this parameter is equal to 0.
* **pathloss\_ht:** this defines the height of the transmitter antenna. for the two-ray ground reflection propagation model. By default, this parameter is equal to 0.5 (m).
* **pathloss\_hr:** this defines the height of the receiver antenna for the two-ray ground reflection propagation model. By default, this parameter is equal to 0.5 (m).

Local Node parameter:

### Shadowing

The library name that can be called in the configuration XML file are:

* shadowing\_none
* shadowing\_lognormal
* shadowing\_mixed

#### Methods Template

They are the methods that should be implemented by a shadowing model.

typedef struct \_shadowing\_methods {

double (\*shadowing) (call\_t \*to\_shadowing, call\_t \*to\_interface, call\_t \*from\_interface, packet\_t \*packet, double rxdBm);

} shadowing\_methods\_t;

#### None

Library name: **shadowing\_none**  
  
Description:  This shadowing model returns a 0 dBm attenuation.

Global Class parameter:

Local Node parameter:

#### lognormal

Library name: **shadowing\_lognormal**  
  
Description:  This shadowing model is a log-Normal shadowing propagation model.

It returns a log-normal random variable for every drawing of the channel with a parameterized deviation.

Global Class parameter:

* **deviation:**standard deviation in dB. Default value is "4"dB.

Local Node parameter:

#### **Mixed Shadowing**

Library name: **shadowing\_mixed**   
  
Description:  This shadowing model is a mixed shadowing propagation model. It could be based on a log-Normal shadowing propagation model with parameters depending on the link condition or none.

The parameter depending on the link condition is the Shadowing Log-Normal standard deviation. For example,  may be equal to 0.5dB, 3dB and 5dB in case of LOS, NLOS and NLOS2 radio link conditions, respectively.

Global Class parameter:

* **Pathloss\_frequency:** this defines the frequency band for the pathloss. Default value is 2400 MHz
* **pathloss\_ht:** this defines the height of the transmitter antenna. for the two-ray ground reflection propagation model. By default, this parameter is equal to 0.5 (m).
* **pathloss\_hr:** this defines the height of the receiver antenna for the two-ray ground reflection propagation model. By default, this parameter is equal to 0.5 (m).
* **shadowing:** this defines the log-normal shadowing model. By default, this parameter is disabled. Possible value for this parameter is: “lognormal”.
* **shadowing\_coherence\_time:** this defines the time during which the lognormal shadowing term is stable or constant. By default, the shadowing coherence time is 500s.
* **shadowing\_deviation\_los:** this defines the log-normal shadowing standard deviation for the case of radio links in LOS condition. By default, this parameter is equal to 0.5 (dB).
* **shadowing\_deviation\_nlos:** this defines the log-normal shadowing standard deviation for the case of radio links in NLOS condition. By default, this parameter is equal to 3 (dB).
* **shadowing\_deviation\_nlos2:** this defines the log-normal shadowing standard deviation for the case of radio links in NLOS2 condition. By default, this parameter is equal to 5 (dB).
* **Symmetry:** this defines in the shadowing is symmetric for each link (source to destination and destination to source). By default, this parameter is equal to 1.

Local Node parameter

### Fading

The library name that can be called in the configuration XML file are:

* fading\_none
* fading\_rayleigh
* fading\_nakagami\_m
* fading\_mixed

#### Methods Template

They are the methods that should be implemented by a fading model.

typedef struct \_fading\_methods {

double (\*fading) (call\_t \*to\_fading, call\_t \*to\_interface, call\_t \*from\_interface, packet\_t \*packet, double rxdBm);

} fading\_methods\_t;

#### None

Library name: **fading\_none**  
  
Description:  This fading model returns a 0 dBm attenuation.

Global Class parameter:

Local Node parameter:

#### Rayleigh

Library name: **fading\_rayleigh**  
  
Description:  This fading model follow a Rayleigh’s law. It returns an attenuation following a Rayleigh random variable for every drawing of the channel.

Global Class parameter:

Local Node parameter:

#### Nakagami

Library name: **fading\_nakagami\_m**  
  
Description:  This fading model follow a Nakagami’s law. It returns an attenuation following a Nakagami random variable for every drawing of the channel. It is computed using the formula:

Global Class parameter:

* **m**: strength of the fading. Default value is "1.0".

Local Node parameter:

#### **Mixed Fading**

Library name:  mixed\_fading  
  
Description:

This fading model is a mixed fadingpropagation model. It could be based on a rice fading propagation model with parameters depending on the link condition or none.

This rice fading model returns an attenuation following a Rice random variable for every drawing of the channel.

The parameter depending on the link condition is the Rice Fading K-factor. For example, may be equal to 9, 5 and 1 in case of LOS, NLOS and NLOS2 radio link conditions, respectively.

Global Class parameter:

* **fading:** this defines the rice fading model. By default, this parameter is disabled. Possible value for this parameter is: “rice”.
* **fading\_k\_los:** this defines the Rice K-factor for the case of radio links in LOS condition. By default, this parameter is equal to 9.
* **fading\_k\_nlos:** this defines the Rice K-factor for the case of radio links in NLOS condition. By default, this parameter is equal to 5.
* **fading\_k\_nlos2:** this defines the Rice K-factor for the case of radio links in NLOS2 condition. By default, this parameter is equal to 1.
* **fading\_coherence\_time:** this defines the time during which the rice fading term is stable or constant. By default, the fading coherence time is equal to 20ms.
* **Symmetry:** this defines in the fading is symmetric for each link (source to destination and destination to source). By default, this parameter is equal to 1.

Local Node parameter:

### Interference

The interference model is responsible for applying interferences of other signals into a given signal. This model is under the umbrella of a PHY models, meaning that it will be called upon request of the PHY model. The PHY model, when necessary, can call the ApplyInterference method passing the tracked signal (signal of interest or the one that will be interfered) and a map with all other signals that may interfere with the tracked one ( interferers). The job of the interference model is to verify how the interferers will affect the signal of interest and update the signal to noise ratio value of the tracked signal, all according to the model created and intended.

The library name that can be called in the configuration XML file are:

* Interference\_none
* Interference\_rf\_signal

#### Methods Template

/\*\* \brief The Abstract Base Class : InterferenceModel Class

\* This means that no instance of the InterferenceModel class can exist.

\* Only classes which inherit from the InterferenceModel class can exist.

\* \fn ApplyInterference() apply the interferences of interferers on the tracked signall

\* \fn Print() prints information about the InterferenceModel

\* \fn GetUID() return the UID of the InterferenceModel

\* \fn ApplyInterferenceImpl() implements the application of interferences on the tracked signal

\* \fn PrintImpl() implements the Print function

\*\*/

class InterferenceModel {

public:

InterferenceModel();

virtual ~InterferenceModel();

void ApplyInterference(std::shared\_ptr<Signal> tracked\_signal, MapOfSignals interferers);

void Print() const;

InterferenceModelUid GetUID() const;

private:

virtual void **ApplyInterferenceImpl**(std::shared\_ptr<Signal> , MapOfSignals) =0 ;

virtual void **PrintImpl**() const;

static InterferenceModelUid uid\_counter\_;

InterferenceModelUid uid\_;

};

#### None

Library name: **interference\_none**  
  
Description:  This interference model should be used when you don’t want any interference being applied to your signals. It basically does nothing to the SINR of the Signal.

Global Class parameter:

Local Node parameter:

#### RF Signal Interference

Library name:  **rf\_signal\_interference**  
  
Description:

We evaluate the Interference between two signals as the amount of energy coming from the interference in a time-frequency proportion of the received packet as follows:

\* psd \*

**∆t**

**∆f**

**Tpacket**

**Interference**

Figure 39: Interference model

is the packet duration of the received packet

is the bandwidth of the interference

is the duration of the interference

is the power spectral density value of the interference

Then, we estimate the **SINR** according to received power (), the noise (N) and interferences of all interferers () as:

Global Class parameter:

Local Node parameter:

### Interferences

This model is for WSNET3.1 kernel. The library name that can be called in the configuration XML file are:

* interferences\_none
* interferences\_orthogonal
* interferences\_factor
* interferences\_factor\_multichannel

#### Methods Template

They are the methods that should be implemented by an interference model.

typedef struct \_interferences\_methods {

double (\* interfere) (call\_t \*to, call\_t \*from, int channel0, int channel1);

} interferences\_methods\_t;

#### : None

Library name: **interferences\_none**  
  
Description:  No interferences occur, even inside a same channel.

Global Class parameter:

Local Node parameter:

#### Orthogonal

Library name: **interferences\_orthogonal**  
  
Description:  The interferences correlation factor is:

* 0: between two different channels.
* 1: inside a channel.

Global Class parameter:

Local Node parameter:

#### Factor

Library name: **interferences\_factor**  
  
Description:  The interferences correlation factor is:

* factor: between two different channels.
* 1: inside a channel.

Global Class parameter:

* **factor**: the correlation factor between two different channels. Default value is 0.5.

Local Node parameter:

#### Factor\_multichannel

Library name: **interferences\_factor\_multichannel**  
  
Description:  The interferences correlation factor is:

* 1: inside a channel.
* factor1: between two adjacent channels. Default value is -49 dB.
* factor2: between channels N and N+2. Default value is -54 dB.
* factor3: between channels N and N+3. Default value is -55 dB.

Global Class parameter:

Local Node parameter:

### Intermodulation

The library name that can be called in the configuration XML file are:

* intermodulation\_none
* intermodulation\_intermodulation

#### Methods Template

They are the methods that should be implemented by an intermodulation model.

typedef struct \_intermodulation\_methods {

double (\* intermod) (call\_t \*to, call\_t \*from, int channel0, int channel1, int channel2);

} intermodulation\_methods\_t;

#### None

Library name: **intermodulation\_none**  
  
Description:  No intermodulation. The intermodulation factor is 0.

Global Class parameter:

Local Node parameter:

#### Intermodulation

Library name: **intermodulation\_intermodulation**  
  
Description:  The intermodulation factor is:

* IMD3: inside a channel.
* IMD3 X factor1: between two adjacent channels. Default value of factor1 is -49 dB.
* IMD3 X factor2: between channels N and N+2. Default value of factor2 is -54 dB.
* IMD3 X factor3: between channels N and N+3. Default value of factor3 is -55 dB.

Where IMD3 Is and IIP3 is the 3rd order interception point.   
Default value of IIP3 is -24dB

Global Class parameter:

Local Node parameter:

### Noise

The library name that can be called in the configuration XML file are:

* noise\_white

#### Methods Template

They are the methods that should be implemented by a noise model.

typedef struct \_noise\_methods {

double (\* noise) (call\_t \*to, call\_t \*from, nodeid\_t node, int channel);

} noise\_methods\_t;

#### White

Library name: **noise\_white**  
  
Description:  This Noise model is a white noise. It returns the noise in mW.

Global Class parameter:

* **white-noise-dBm** : Noise in dBm. Default value is 0 mW i.e. MIN\_DBM dBm.

Local Node parameter:

### Energy

The library name that can be called in the configuration XML file are:

* Energy\_dummy\_energy
* Energy\_linear

#### Methods Template

They are the methods that should be implemented by an energy model.

typedef struct \_energy\_methods {

void (\*recharge) (call\_t \*to, double energy);

void (\*consume) (call\_t \*to, double current, uint64\_t duration);

double (\*energy\_consumed) (call\_t \*to);

double (\*energy\_recharged) (call\_t \*to);

double (\*energy\_remaining) (call\_t \*to);

double (\*energy\_status) (call\_t \*to);

double (\*get\_supply\_voltage) (call\_t \*to);

} energy\_methods\_t;

These methods are linked in model\_handler/energy.c&h

#### Dummy

Library name: **energy\_dummy\_energy**  
  
Description:  This energy model is a dummy battery i.e. without battery decrease.

Global Class parameter:

Local Node parameter:

#### Linear Battery

Library name: linear\_battery  
Description: This energy model is a linear battery i.e. with a linear battery decrease.

Global Class parameter:

* **global\_consumption\_J**: defines the global consumption in joules (J). Default value is "0".

Local Node parameter:

* **available\_energy\_J:**the current available energy in the battery in joules (J) (1mWh = 3.6J). Default value is " -1".
* **battery\_capacity\_mAh:**the maximum capacity of the battery in milliampere-hour(mAh). Default value is " 1000".
* **battery\_capacity\_J:** the initial energy capacity of the battery in joules (J). The initial value is calculated based on the value of battery\_capacity\_mAh.
* **consumed\_energy\_J:** the energy that has been consumed from the battery in joules (J). The initial value is “0”.
* **recharged\_energy\_J:** the energy that has been recharged to the battery in joules (J). The initial value is “0”.
* **voltage\_V:**the voltage used by battery in volts (V). Default value is "3"

### Mobility

The library name that can be called in the configuration XML file are:

* Mobility\_dummy\_mobility
* Mobility\_static
* Mobility\_filestatic
* Mobility\_billiard
* Mobility\_torus\_plane
* Mobility\_torus\_central
* Mobility\_teleport
* Mobility\_static\_wiplan
* Mobility\_position\_inside\_room

#### Template

They are the methods that should be implemented by a mobility model.

typedef struct \_mobility\_methods {

void (\*update\_position) (call\_t \*to, call\_t \*from);

double (\*get\_speed) (call\_t \*to);

angle\_t (\*get\_angle) (call\_t \*to);

} mobility\_methods\_t;

#### Dummy

Library name: **mobility\_dummy\_mobility**  
  
Description:  This mobility model is a dummy model. Each node starts with a random (x,y,z) position or a position defined in the XML configuration file. At each update, the node moves to (x+1, y+1, z+1) position.

Global Class parameter:

Local Node parameter:

* **X** : Initial abscissa position in meter. Default value is random in the topology area.
* **Y** : Initial ordinate position in meter. Default value is random in the topology area.
* **Z** : Initial height position in meter. Default value is random in the topology area.

#### Static

Library name: **mobility\_static**  
  
Description:  This mobility model is a model for no mobility. Each node starts with a random (x,y,z) position or a position defined in the XML configuration file.

Global Class parameter:

Local Node parameter:

* **X** : Initial abscissa position in meter. Default value is random in the topology area.
* **Y** : Initial ordinate position in meter. Default value is random in the topology area.
* **Z** : Initial height position in meter. Default value is random in the topology area.

#### Filestatic

Library name: **mobility\_filestatic**  
  
Description:  This mobility model is a model for no mobility. Each node starts with a (x,y,z) position imported from a file.

Global Class parameter:

* **file:** file containing the node positions. Default value is "mobility.data".

Local Node parameter:

#### Static\_wiplan

Library name: **mobility\_static\_wiplan**  
  
Description:  This mobility model is another model for no mobility where each node starts with a (x,y,z) position imported from a file (wiplanout.xml).

Global Class parameter:

* **file:** file containing the node positions. Default value is " wiplanout.xml ".

Local Node parameter:

#### Billiard

Library name: **mobility\_billiard**  
  
Description:  This mobility model is a billiard like mobility model. Each node starts with a random (x,y,z) position or a position defined in the XML configuration file. At each update, the node moves to the next position according to the randomly selected speed and randomly selected angle. In this model, nodes move and rebound against the network boundaries.

Global Class parameter:

* **max-speed:**node maximum speed in m/s used when a node speed is randomly chosen. Default value is "30"m/s.

Local Node parameter:

* **x:**x coordinate of the node. Default value is "random".
* **y:**y coordinate of the node. Default value is "random".
* **z:**z coordinate of the node. Default value is "random".
* **angle-xy:**node direction in the xy plane. Default value is "random".
* **angle-z:**node direction regarding the xy plane. Default value is "random".
* **speed:**node speed in m/s. Default value is "random" but lower than max-speed.

#### Torus Central

Library name: **mobility\_torus\_central**  
  
Description:  This mobility model is a torus central like mobility model. Each node starts with a random (x,y,z) position or a position defined in the XML configuration file. At each update, the node moves to the next position according to the randomly selected speed and randomly selected angle. The angle and the speed stay constant during the whole simulation. In this model, when a node exits the network area, it re-enters at the symmetrical position regarding central symmetry.

Global Class parameter:

* **max-speed:**node maximum speed in m/s used when a node speed is randomly chosen. Default value is "30"m/s.

Local Node parameter:

* **x:**x coordinate of the node. Default value is "random".
* **y:**y coordinate of the node. Default value is "random".
* **z:**z coordinate of the node. Default value is "random".
* **angle-xy:**node direction in the xy plane. Default value is "random".
* **angle-z:**node direction regarding the xy plane. Default value is "random".
* **speed:**node speed in m/s. Default value is "random" but lower than max-speed.

#### Torus Plane

Library name: **mobility\_torus\_plan**  
  
Description:  This mobility model is a torus plan like mobility model. Each node starts with a random (x,y,z) position or a position defined in the XML configuration file. At each update, the node moves to the next position according to the randomly selected speed and randomly selected angle. The angle and the speed stay constant during the whole simulation. In this model, when a node exits the network area, it re-enters at the symmetrical position regarding plane symmetry.

Global Class parameter:

* **max-speed:**node maximum speed in m/s used when a node speed is randomly chosen. Default value is "30"m/s.

Local Node parameter:

* **x:**x coordinate of the node. Default value is "random".
* **y:**y coordinate of the node. Default value is "random".
* **z:**z coordinate of the node. Default value is "random".
* **angle-xy:**node direction in the xy plane. Default value is "random".
* **angle-z:**node direction regarding the xy plane. Default value is "random".
* **speed:**node speed in m/s. Default value is "random" but lower than max-speed.

#### Teleport

Library name: **mobility\_teleport**  
  
Description:  This mobility model is a teleport mobility model. Each node starts with a random (x,y,z) position or a position defined in the XML configuration file. It waits during a pause\_time (random duration between min-pausetime and max\_pausetime or defined in the XML configuration file) and teleports itself to a new random position. Then it repeats the process.

Global Class parameter:

Local Node parameter:

* **x:**x coordinate of the node. Default value is "random".
* **y:**y coordinate of the node. Default value is "random".
* **z:**z coordinate of the node. Default value is "random".
* **pausetime:**node pause time. Default value is "2s".
* **min-pausetime:**minimum node pause time. Default value is "0s".
* **max-pausetime:**maximum node pause time. Default value is "0s".

#### Position inside room

Library name: **mobility\_position\_inside\_room**  
  
Description:  This mobility model is a model to be used along indoor\_map. Each node starts with a random (x,y,z) position delimitated by each of the defined rooms so that a number of nodes are randomly displaced inside each room. The user can also set a fixed position defined in the XML configuration file. After that, the node remains on the same position throughout the simulation.

Global Class parameter:

Local Node parameter:

* **x\_value** : Initial abscissa position in meter. Declare this value only you need to explicitly put a node in a given position.
* **y\_value** : Initial ordinate position in meter. Declare this value only you need to explicitly put a node in a given position.
* **nb\_node\_per\_room :** defines the number of nodes will be inside each room. Default value is “1”.
* **room\_width :** defines the room width. This value should be the same as the one declared on the indoor\_map
* **room\_length :** defines the room length. This value should be the same as the one declared on the indoor\_map

### Global map

The library name that can be called in the configuration XML file are:

* Global\_map\_basic

#### Methods Template

They are the methods that should be implemented by a global map model.

typedef struct \_global\_map\_methods {

position\_t \* (\*get\_global\_area) (call\_t \*to, call\_t \*from);

} global\_map\_methods\_t;

#### Basic

Library name: **global\_map\_basic**  
  
Description:  This Global\_map model is a basic global map model defining the area of the simulation. It returns the upper-right corner position.

Global Class parameter:

* **X** : Abscissa position in meter. No default value thus must be configured.
* **Y** : Ordinate position in meter. No default value thus must be configured.
* **Z** : Height position in meter. No default value thus must be configured.

Local Node parameter:

### Map

The library name that can be called in the configuration XML file are:

* map\_dummy\_map
* map\_indoor\_map

#### Methods Template

They are the methods that should be implemented by a map model.

typedef struct \_map\_methods {

int (\*get\_scenario\_condition) (call\_t \*to, call\_t \*from, nodeid\_t src, nodeid\_t dst);

int (\*get\_nbr\_walls) (call\_t \*to, call\_t \*from, nodeid\_t src, nodeid\_t dst);

} map\_methods\_t;

#### Dummy

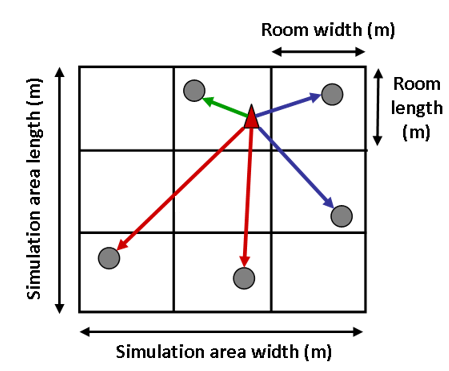
Library name: **map\_dummy\_map**  
  
Description:  This map model is a basic map model.

Global Class parameter:

Local Node parameter:

#### Indoor map

Library name: **map\_indoor\_map**  
  
Description:  This map model describes a basic indoor map containing several rooms of rectangular shape, all of them with the same width and height. The model will print in the beginning of the simulation the following information:



[MAP] size\_x=180 size\_y=180 room\_wid = 20 room\_len = 20 [nr\_room\_x = 9 nr\_room\_y = 9 nb\_room\_total = 81]

This are the described and calculated values of the size of the whole simulation are on both axes, the room width and length of every room and the number of room on the X and Y axes as well as the total number of rooms.

Global Class parameter:

* **room\_width:**The room or subarea width (indoor). Default value is equal to the maximum value of the X axis of the whole simulation area.
* **room\_length:** The room or subarea height (indoor). Default value is equal to the maximum value of the Y axis of the whole simulation area.

Local Node parameter:

### Monitor

The library name that can be called in the configuration XML file are:

* Monitor\_dummy\_monitor

#### Methods Template

They are the methods that should be implemented by a monitor model.

typedef struct \_monitor\_methods {

void (\*fire) (call\_t \*to, call\_t \*from, void \*data);

} monitor\_methods\_t;

#### Dummy

Library name: **monitor\_dummy\_monitor**  
  
Description:  This monitor model is a basic monitor model.

Global Class parameter:

Local Node parameter:

### Sensor

The library name that can be called in the configuration XML file are:

* Sensor\_dummy\_sensor

#### Methods Template

No methods template is available for sensor.

#### Dummy

Library name: **dummy\_sensor\_sensor**  
  
Description:  This sensor model is a basic sensor model.

Global Class parameter:

Local Node parameter:

### Physical

The library name that can be called in the configuration XML file are:

* Physical\_dummy\_physical

#### Methods Template

No methods template is available for physical.

#### dummy

Library name: **physical\_dummy\_physical**  
  
Description:  This physical model is a basic physical environment model.

Global Class parameter:

Local Node parameter:

### Link

The link defines a set of services that describes the state and characteristics of the different communication links established between interfaces throughout the simulation. They can depend on the orientation of both sender and receiver, the position on the map, etc.

#### Methods Template

They are the methods that should be implemented by a link model.

typedef struct \_link\_methods {

int (\*get\_link\_condition) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

int (\*get\_link\_type) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

int (\*get\_communication\_type) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

double (\*get\_mutual\_orientation) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

int (\*get\_complementary\_link\_condition) (call\_t \*to\_link, call\_t \*to\_interface, call\_t \*from\_interface);

} link\_methods\_t;

#### dummy

Library name: **link\_dummy\_link**  
  
Description:  This module defines a dummy link, by default with the type of radio link LOS (Line-Of-Sight).

Global Class parameter:

Local Node parameter:

#### Indoor link

Library name: **link\_indoor\_link**  
  
Description:  This module defines the type of radio link LOS (Line-Of-Sight), NLOS (Non-Line-de-Sight) and NLOS2 (non-line-of-Sight 2) according to the map in which the nodes will be deployed.

The sensor field is modeled as a two-dimensional simulation area in which wireless nodes (sensors, sinks and anchors) are uniformly and randomly deployed. Depending on the considered application scenario (indoor or outdoor), this area may be divided into a set of equal-size subareas (or rooms) to account for the impact of the environment (e.g. presence of walls, obstacles, etc.) on the propagation of radio signals.

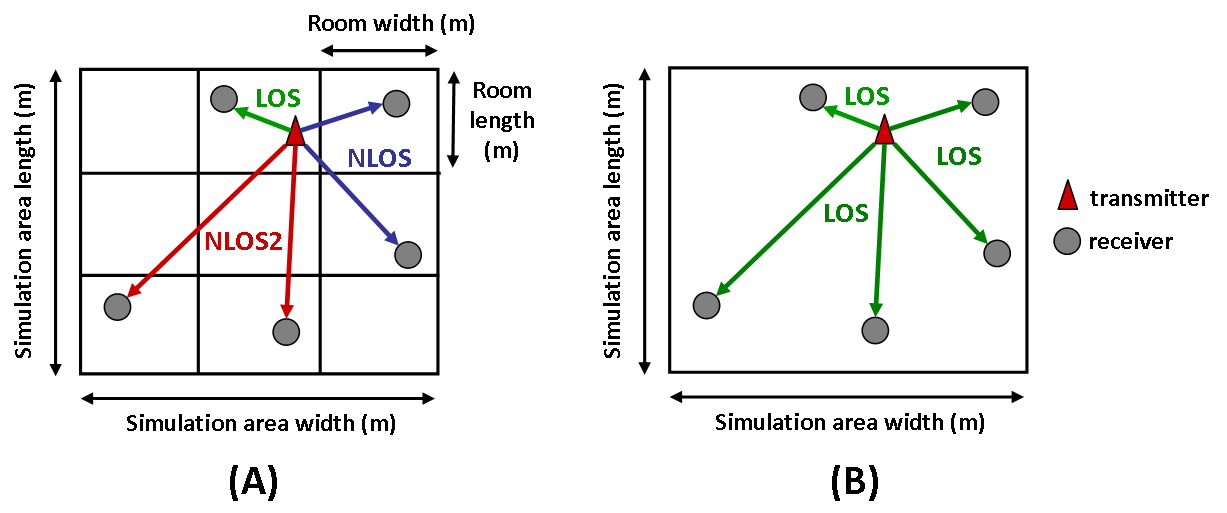


Figure 40: « indoor\_map » model description

The simulation area may be divided into equal-size subareas to model realistic indoor environments (in the above example, the environment has 9 rooms). Hence, depending on the location of the wireless nodes inside the sensor field, the corresponding radio links are spatially correlated and can be considered as:

* In ***LOS condition***: if the receiver and the transmitter are located in the same room
* In ***NLOS condition***: if the receiver and the transmitter are located in immediately adjacent rooms
* In ***NLOS2 condition***: if the receiver and the transmitter are located in non-adjacent rooms
* In ***NLOS3 condition***: if the receiver and the transmitter are located in far-away rooms

When considering outdoor environments, the overall simulation area is not divided into subareas. In this case, all radio links are considered as being in direct Line-Of-Sight (LOS) conditions.

Global Class parameter:

Local Node parameter: