## Version History and Installation

### Version 2.0

The first online version of WSNet is available in the following website: <http://wsnet.gforge.inria.fr/download.html>

One of the key advantages of using WSNet 2.0 simulator is that there is an associated node platform simulator WSIM, which allows simulating different components of the sensor nodes. WSIM is available in the following website <http://wsim.gforge.inria.fr>

#### System requirements

The WSNet simulator can only be installed on a Linux-based operating system (OS), including RedHat, Debian, Ubuntu, Fedora, Suse, Mandrake, MAC OS X, and so on. Before compiling WSNet make sure the following software packages are installed on your Linux system:

* gcc, libtool and make
* autoconf (>= 2.61), and pkg-config (>= 0.21)
* sun-java5-jdk, or openjdk-6-jdk (for graphical tools)
* libglib2.0-0, and libglib2.0-dev
* libxml2, and libxml2-dev
* libgsl0-dev

For example, by considering an Ubuntu or Debian based Linux operating system, the following linux command should install all the required software packages:

**sudo apt-get install glibc subversion bash doxygen gcc libtool make autoconf pkg-config sun-java5-jdk libglib2.0-0 libglib2.0-dev libxml2 libxml2-dev libgsl0-dev**

It should be noted that depending on the Linux version and family name (Redhat, Ubuntu, Suse, Fedora, etc.), the names as well as the versions of the above packages may change from one linux system to another. Furthermore, a root-access on the Linux operating system is not necessary to download, install, compile and use the WSNet simulator

#### Downloading WSNet source code

Check out the very latest WSNet source code from the SVN repository by typing the following command:

**$**  **svn checkout svn://scm.gforge.inria.fr/svn/wsnet**

The previous linux command will download the WSNet source code from the official SVN repository, and a new directory named wsnet will be created in the home directory (i.e. /usr/local/ wsnet/).

By default, the WSNet source code directory (i.e /usr/local /wsnet/) is organized according to the following main subdirectories:

* **demo/:** examples of XML configuration files to run simulations;
* ***include/***: the WSNet core and API shared headers (\*.h);
* ***libraries/***: the WSNet core dynamic libraries for memory management, etc.
* ***models/***: contains the source code of all the WSNet simulation modules:
  + ***models/antenna***/: antenna models (omnidirectionnal, etc.) ;
  + ***models/energy***/: battery models (linear, etc.);
  + ***models/interference***/: interference models (factor, orthogonal, etc.);
  + ***models/modulation***/: modulation models (bpsk, o-qpsk, etc.);
  + ***models/noise***/: noise models (white, etc.);
  + ***models/radio***/: radio models (802.15.4, generic radio, etc.);
  + ***models/application***/: application models (cbr, hello, etc.);
  + ***models/environment***/: environment models (fire, etc.);
  + ***models/mac***/: MAC models (BMAC, 802.11, 802.15.4, etc.);
  + ***models/mobility***/: mobility models (static, billiard, etc.);
  + ***models/monitor***/: monitoring models (energy, noise, etc.);
  + ***models/propagation***/: propagation models (freespace, etc.);
  + ***models/routing***/: routing models (greedy, static, etc.);
* **src/:** contains the source code of the core of the WSNet simulator;
* ***user\_models/***: contains the files needed to implement new external modules;
* ***utils/***: contains some useful matlab / BASH shell scripts, etc.

#### Compiling WSNet

After checking out the SVN repository, move to the ./wsnet/ directory and type the following commands:

**$**  **./bootstrap**

**$**  **./configure**

The main configure options are (type ./configure --help for a more complete list):

* --prefix=PREFIX: to specify an -2.0/
* --enable-debug: enable debugging symbols
* --enable-optimize-nodes: optimize node management with spatial data structures
* --enable-periodic-mobility: set periodic mobility instead of event driven
* --with-mem-fs=policy: fixed size memory allocation policy (default policy is **prealloc**, other possibility is **malloc**)
* --with-das=structure: unsorted data structure (default structure is a **list**)
* --with-sodas=structure: sorted data structure (default structure is a **heap**, other possibility is a **list**)
* --with-spadas=structure: spatial data structure (default structure is **grid**, other possibility is **flat** and **dbtree**)
* --with-hadas=structure: hashed data structure (default structure is a **hash** table of lists)
* --with-scheduler=support: scheduler support (default support is local and distant nodes, other possibility is **local**)

To compile the WSNet source code, type the following command:

**$**  **make**

#### Installing WSNet

In order to install WSNet, type the following command (with root privileges):

**#**  **make install**

By default, WSNet will be installed at the /usr/local/wsnet-2.0/ directory. The install directory is organized as follows:

* /usr/local/wsnet-2.0/bin/: simulator binary files (wsnet, wsnet-replay, wsnet-topogen, ...)
* /usr/local/wsnet-2.0/lib/: shared libraries for application, routing, mac and radio protocols
* /usr/local/wsnet-2.0/shared/: shared resources
* /usr/local/wsnet-2.0/include/: header files for implementing new modules
* /usr/local/wsnet-2.0/demo/: demo simulation files

You can add the WSNet install directory in the search path by adding the following entries in ~/.bashrc:

PATH=$PATH:/usr/local/wsnet-2.0/bin   
export PATH

#### Usage

Once WSNet is properly compiled and installed you can run a simulation by executing the wsnet binary file. Possible options are:

* -c configfile.xml: to specify a config file for the simulation parameters
* -S rng-seed: to specify a seed for the random number generator
* -R rng-type: to specify the random number generator type. Possible value are: mt19937, ranlxs0, ranlxs1, ranlxs2, ranlxd1, ranlxd2, rng\_ranlux and rng\_ranlux389. The default value is mt19937.
* -V : to print the version number

For running a simulation using the cbr.xml config file, one would type the following command:

**#**  **wsnet -c /usr/local/wsnet-2.0/demo/cbr.xml**

### Version 3.0

WSNet is a simulation tool for wireless networks. It allows to do technical choices about a network and its nodes before the deployment of it. WSNet 3.0 is a refactoring of WSNet 2.0.

The entire source code is CeCILL FREE v2.1. It is a French Free Software license, compatible with the GNU GPL. <http://www.cecill.info/index.en.html>

The main modifications are:

* Reorganization of source code
* Decorrelation of medium and environment. Several mediums and environments are possible.
* Modification of the file parser
* Modification of call-t structure
* Addition of an example with multi-mediums

#### System requirements

Before compiling WSNet 3.0 make sure the following software packages are installed on your Linux system:

* Gcc
* Libtool
* Make
* Sun-java5\_jdk
* libglib2.0-0
* libglib2.0-dev
* libxml2
* libxml2-dev
* libgsl0-dev
* automake
* autoconf

#### Downloading WSNet source code

Check out the very latest WSNet 3.0 source code from the SVN repository by typing the following command:

**$**  [**svn://scm.gforge.inria.fr/svnroot/wsnet-3/trunk**](svn://scm.gforge.inria.fr/svnroot/wsnet-3/trunk)

#### Configure WSNet 3.0

After checking out the SVN repository, move to the ./wsnet/ directory and type the following commands:

**$**  **./bootstrap**

**$**  **./configure**

To compile the WSNet source code, type the following command:

**$**  **make**

In order to install WSNet, type the following command (with root privileges):

**#**  **sudo make install**

By default, WSNet will be installed at the /usr/local/wsnet-3.0/ directory.

You can add the WSNet install directory in the search path by adding the following entries in ~/.bashrc:

export PATH=$PATH:/usr/local/wsnet-3.0/bin

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/wsnet-3.0/lib

### Version 3.0 (HarvWSNet)

HarvWSNet is a new version of WSNET 3.0 including a cosimulation framework that supports any type of energy harvesting technology, energy management algorithm and energy storage element. It associates WSNet, an open-source discrete-events WSN simulator, with Matlab, the most commonly employed tool in the field of energy harvester modelling. WSNet and Matlab exchange data using TCP sockets that are accessible by each node’s energy module. The simulation time is controlled by WSNet.

The goal of HarvWSNet is to evaluate the power charge/discharge profile of energy harvesting nodes in realistic scenarios. To this end, the simple battery model of WSNet is replaced by a function which calls directly the energy harvesting system implemented in MATLAB. The global architecture of the proposed framework is represented in Figure 19.

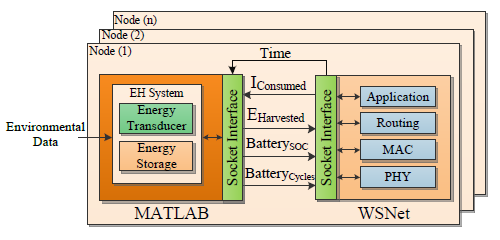


Figure 19: Structure of HarvWSNet

Each wireless node is modeled by a distinct stack of protocol elements (PHY, MAC, Routing, Application, etc.) and possesses an energy module (with or without harvesting capabilities) that is modeled in Matlab. Each node can potentially have a distinct type of energy harvester.

In the co-simulation process, both MATLAB and WSNet operate in parallel and interactively. A simple synchronization process between these tools was implemented in HarvWSNet which requires only a single computer. The simulation is controlled by WSNet which defines the master clock and is responsible for clock synchronization.

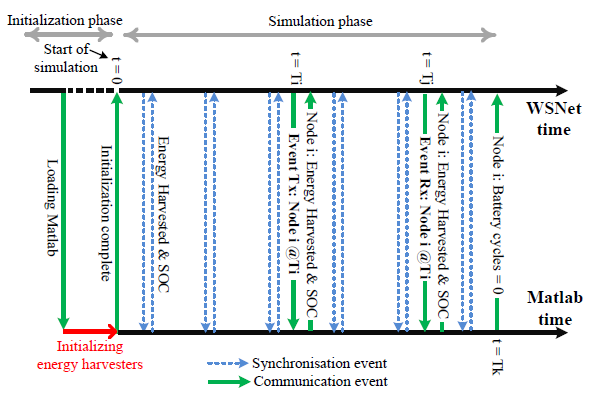


Figure 20: Interactive co-simulation of WSNet and MATLAB.

Figure 20 shows the interactions between WSNET and Matlab during the co-simulation process. An energy harvester is a dynamic system in which the battery state-of-charge (SOC) and energy harvested change in a continuous manner with respect to time. On the other hand, WSNet is a discrete-event driven simulator that handles events that are unevenly distributed in time due to the stochastic nature of packet generation. We therefore define two types of events at which WSNet invokes Matlab. The first, called synchronization events, are scheduled at fixed intervals and dedicated to maintaining the synchronization of the simulation clocks between WSNet and Matlab. The second type of events, called communication events, arise when a node is in one of the following states: transmitting, receiving and listening, or in any other power consuming state. The Matlab model is updated whenever WSNET detects an energy consuming task (e.g. data transmission), an energy harvesting event (e.g. a car passage that produces wind-flow that drives a micro-wind turbine) or when a specific synchronization event occurs. Matlab returns the battery state-of-charge and remaining number of battery recharge cycles (when appropriate) which will limit the node lifetime. Due to the modeling power of Matlab, this approach has the advantage of allowing for extremely complex and complete energy harvester models.

During the initialization phase which takes place at the initial time but before the actual simulation starts, WSNet creates the nodes specified in the simulation configuration file and requests a connection to the Matlab engine via sockets. Once the connection is setup, Matlab initializes the energy harvesting system that corresponds to each simulated node and associates the predefined environmental data in accordance with the location of each node. After initialization, WSNet begins the simulation at time zero and passes the clock to Matlab each T seconds allowing the energy harvester of each node to calculate the scavenged power, the battery SOC and the number of remaining charge/discharge cycles. The period T is controlled by WSNet and is chosen as a function of the communication duty-cycle and the environmental data type. When a node transmits or receives a packet, WSNet sends a notification to Matlab and waits until the Matlab calculation is done before continuing. The notification includes the node number, the kind of event (e.g. reception or transmission), the event duration, and the current to draw from the battery. Otherwise, if the node is not receiving nor transmitting data, the energy harvesting system considers that the node is in idle mode and draws the current corresponding to this state. When the battery associated to a node reaches its maximum number of charge/discharge cycles, Matlab sends a notification to WSNet indicating that the node is not operational and should be removed from the network.

### Version 3.1

WSNET simulator has been adapted to BAN and BBN contexts in order to fulfill the upper layers needs (i.e. MAC, Network and application layer) with respect to the multi-scale dynamics of Inter-BAN multi-link channels.

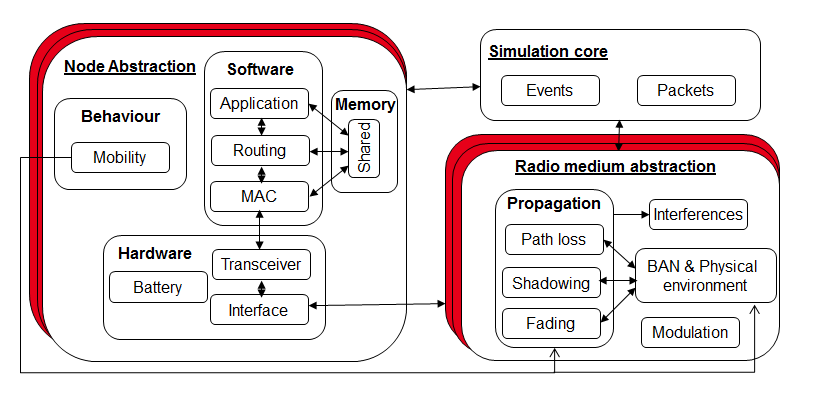


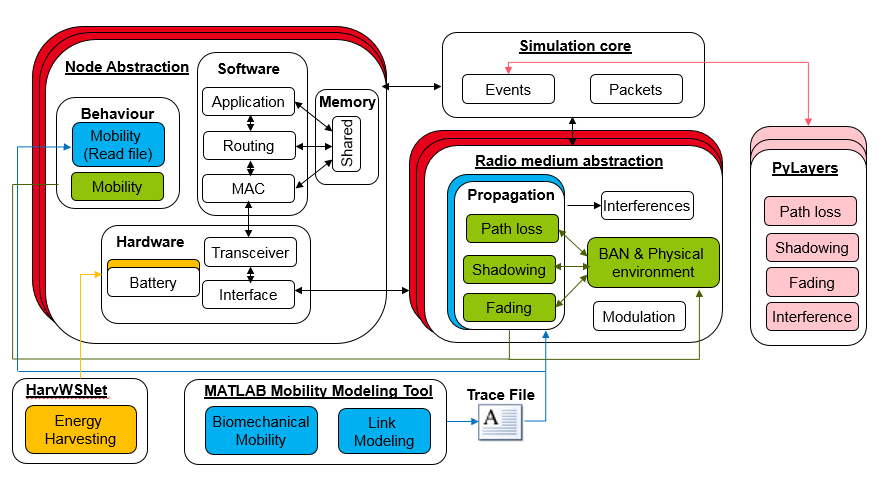
Figure 21: Adaptation of WSNet simulator to BAN/BBN Networks

The BAN & Physical environment module exchanges information with the mobility model (e.g. positions, directions, speeds, obstruction) and with BAN propagation models (e.g. link conditions) in order to select the corresponding channel model in our database.

An essential feature of our simulation is our semi-deterministic approach instead of statistical approach. The mobility model provides for each realization some important deterministic information (i.e. positions, relative orientations, speeds, obstructions...) which are treated as input parameters by Body-to-Body and On-Body channel models to generate realistic traces (i.e. with consistent mobility patterns) as a function of time.

For example, On-Body channels can adapt their models based on human walking according to the BAN speed. Body-to-Body channels can dynamically take into account BAN & Physical environment information (e.g. the mutual orientations, the (Non) Line-Of-Sight conditions and the room the other BAN belongs) for interfering links.

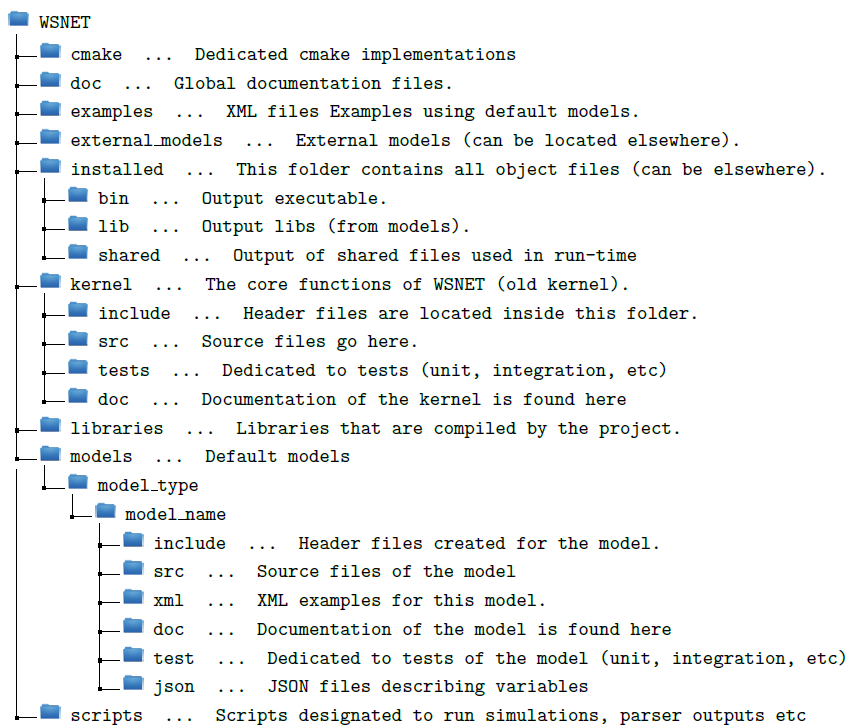
Moreover, a Physical simulator PyLayers has been interfaced in real time with WSNet. This open source project addresses indoor radio propagation and provides a site-specific Ultra Wide Band oriented simulator for BANs. The idea is to simulate on real time the PyLayers mobility feature with statistical radio channel models (PHY layer) and WSNet (upper layers) calculating the performance of the simulated scenario. The simulator embeds several modules including: an agent mobility simulator based on discrete event simulator to which a multi-cylinder body model can be associated; a deterministic channel propagation simulator based on ray tracing approach for Body-to-Body and Off-Body communications; a statistical model for On-Body communications taking into account the environment and the body shadowing effects.



**Figure 22:** **Actual architecture of WSNet 3.0 including HarvWsnet interaction, BAN adaptation, pylayers interface and Matlab interface through trace file.**

Another partner has interfaced WSNet with MATLAB using trace file to introduce new mobility and propagation models.

New features have been added in WSNET 3.1:

* New code architecture:   
    
    
  **Figure 23:** **Current code architecture of WSNET.**
* Cmake compilation architecture: CMake now searches inside models folder and identify automatically all available model types along with the models defined in each model type folder. In this way, we can create new repositories with a different number of models available and CMake will be able to identify them automatically, without cumbersome changes on the code.
* Simplification on how to configure and compile WSNET by using CMake.
* Add template for each model types and simplification how to create new external models
* Creation of link models.
* Creation of group architecture to deal with groups of nodes with similar behavior.
* Clean absolete models and kernel
* Fix some memory leaks
* Use of CMake to create tests that are further run by CTest.
* Add integration test with ctest for models in core. Some Xml files are used to test each model separately. We rely on a minimum set of models that are required to be error-free, once we need several models to run WSNET. The models today are:
  + fading : none
  + pathloss : none
  + shadowing : none
  + interfernces : none
  + intermodulation : none
  + noise : white
  + modulation : oqpsk
  + map : dummy\_map
  + global\_map : basic
  + application: demo
  + mac : 802\_15\_4\_u\_csma\_ca\_2400\_oqpsk
  + radio : radio\_802\_15\_4\_2400\_oqpsk
  + interface : antenna\_omnidirectionnal
  + mobility : static
* Add Jenkins tests for cppcheck tests, valgrind tests
* Improve energy and transceiver models

#### System requirements

Before compiling WSNet 3.1 make sure the following software packages are installed on your Linux system:

* Libtool
* Make
* CMake
* libglib2.0-0
* libglib2.0-dev
* libgmodule2.0
* libxml2
* libxml2-dev
* libgsl0-dev

#### Downloading WSNet source code

The WSnet source code has been package (WSNET\_source\_code\_v31) and could be delivered on demand.

#### Environment variable

In order to configure and properly compile WSNET we need to define a number of environment variables. Below, you can find the lines added to the .cshrc file, you should consider doing the same.

# Path configurations of WSNET

setenv WSNET\_SRC\_PATH /path/to/wsnet/folder/

# Path of the installed resources

setenv WSNET\_INSTALLED\_PATH $WSNET\_SRC\_PATH/installed

# Path of the external models

setenv WSNET\_EXTERNAL\_MOD\_PATH $WSNET\_SRC\_PATH/external\_models

# Paths to the external libs after installation

setenv WSNET\_MODDIR $WSNET\_EXTERNAL\_MOD\_PATH/lib

# General PATH Variable

set path = ($WSNET\_INSTALLED\_PATH/bin/ $path)

Note that you need to change the /path/to/wsnet/folder/ to the folder on which you have downloaded WSNET. Moreover, you can also change the destination folder for the installation. Simply change the folder of WSNET\_INSTALLED\_PATH. The same is valid for external models (WSNET\_EXTERNAL\_MOD\_PATH).

#### Configure and compile WSNet 3.1

In order to compile WSNET, you'll need a build folder. This folder is used to keep source files separated from the CMake and Make generated files. In this way, we do not pollute our source files with building files.

If you still don't have it, create the build folder before proceeding:

**$ cd $WSNET\_SRC\_PATH**

**$ mkdir build**

Now that you have already created your build folder, you can access it:

**$ cd build**

In the build folder, you will be able to compile WSNET.

**$ cmake ..**

CMake will then configure, check all dependencies and generate the Makefiles that will be used to compile. The output will be something similar to:

**-- The C compiler identification is GNU 4.1.2**

**-- The CXX compiler identification is GNU 4.1.2**

**-- Check for working C compiler: /usr/bin/cc**

**-- Check for working C compiler: /usr/bin/cc -- works**

**-- Detecting C compiler ABI info**

**-- Detecting C compiler ABI info - done**

**-- Detecting C compile features**

**-- Detecting C compile features - done**

**-- Check for working CXX compiler: /usr/bin/c++**

**-- Check for working CXX compiler: /usr/bin/c++ -- works**

**-- Detecting CXX compiler ABI info**

**-- Detecting CXX compiler ABI info - done**

**-- Detecting CXX compile features**

**-- Detecting CXX compile features - done**

**-- Checking for ccache**

**-- Configuring and generating files for Linux**

**-- Found PkgConfig: /usr/bin/pkg-config (found version "0.21")**

**-- Checking for one of the modules 'gmodule-2.0'**

**-- Found GSL: /usr/include (found version "1.13")**

**-- Found LibXml2: /usr/lib64/libxml2.so (found version "2.6.26")**

**-- Found M: /usr/lib64/libm.so**

**-- Configuring done**

**-- Generating done**

**-- Build files have been written to: /path/to/wsnet/folder/build**

If cmake is not found, you can add an alias in .cshrc file with its location:

alias cmake /home/prog/cmake/cmake-3.6.2/bin/cmake

If no errors were produced, you can compile the code:

**$ make all**

If you want to install the files in the dedicated folder (WSNET\_INSTALLED\_PATH), you can proceed to:

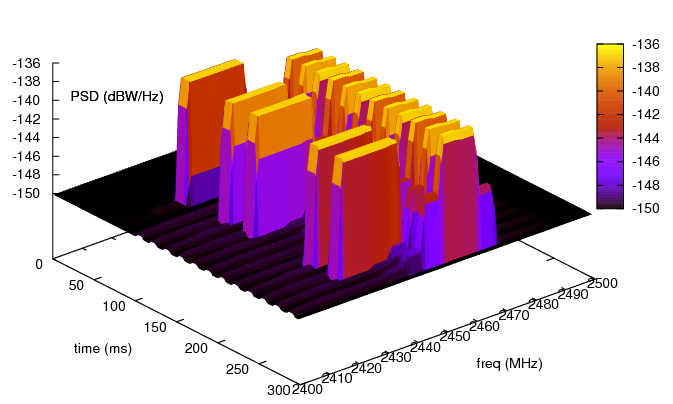
**$ make install**

After this, you will be able to run wsnet from the command line.

### Version 4.0

WSnet 4.0 is now written in C and Modern C++. WSNET simulator has been extended in several aspects (e.g., spectrum use, interference, capture effect) to take into account flexibility and specificity of PHY/MAC layers. The core of WSNet simulator has been modified to support massive scenarios and to support the PHY layer heterogeneity and flexibility.

To support massive scenarios, the kernel of the simulator has been improved in ModernC++. The new core includes a spectrum model in order to provide a support for modeling the frequency-dependent aspects of communications:



***Figure 24:******Spectrum model of Baldo extended for WSNet (as an illustrative example)***

* Exploit the spectrum in terms of spectral resource instead of logical channel
* More accurate PHY models (Several Waveforms with different configurations)
* More accurate interference model for heterogeneous simulations
* Allow non-contiguous Channel Bonding simulations

It is important to notice that all layers above transceiver have full backward compatibility. Thus, packets can be generated and used as before without need of redesigning. However, mind that this may change on the near future as the packet\_t type has several unused members that are now obsolete. Moreover, to take full advantage of the new features added to the simulator, we need to update this layers so that they orchestrate better with the lower layers. Furthermore, a transceiver model from a 3.0 version needs little to none change to work currently, however not taking the full potential uses of the new features. A packet typically created on the application layer is transmitted to the lower layers as it was done on the 3.0 version. When the packet arrives on the interface (antenna), it then forward the packet to the PHY model which received the packet on its ReceiveFromUp\_Deprecated interface (API) call. It is “Deprecated” as the packet still contains several member which were very useful for the previous versions but are no longer necessary. A cleaning on the packet\_t structure (or maybe the creation of a new one) is needed. The PHY will then create a Signal (updating the PSD, time, frequency) and insert the packet inside.

WSNET4.0 simulations need specific models: PHY, Spectrum, Signal tracker, Interference.

New features have been added in WSNET 4.0:

* New code architecture
* Include Modern C++ for WSNet Core
* Addition of models (spectrum, PHY, signal tracker, interference,…)
* Clean obsolete models and kernel
* Fix some memory leaks
* Improvement of Cmake compilation architecture
* Use of CMake to create tests that are further run by CTest.
  + Add continuous integration practices
    - Add tests for cppcheck, valgrind
    - Add unit test
    - Add integration test with ctest for models in core.
* Add support to C++ models in WSNET (with wrappers to provide clean declaration and dynamic linkage)
  + Add Abstract Classes to Models (along with their factories) described before: (signal, waveform, intervals, spectrum, phy, interference, )
  + Add data structures (interval search tree)
  + Add implementation of some models
  + Add glue code to call new models instead of classical ones

#### System requirements

WSNet has been used on the following platforms:

* Linux
* Mac OS X (partially)

These are the base requirements to build and use WSNet from a source:

* [CMake](https://cmake.org/) v3.6.2 or newer
* GNU-compatible Make
* POSIX-standard shell
* GCC v4 or newer
* [Python](https://www.python.org/) v2.8 or newer (for running some of the tests and scripts)

Libraries:

* libxml2
* libgmodule2.0

Note that if you are planning to contribute code, you'll need the development files of the libraries (e.g. libxml2-dev).

#### Downloading WSNet source code

The kernel of WSNET 4.0 can also be found on GITHUB

<https://github.com/CEA-Leti/wsnet>

#### Environment variable & Configure and compile WSNet 3.1

See version 3.1 in sections 1.3.4.3 and 1.3.4.4

## Compilation

### Cmake Build Types

CMake Build Types statically specifies what build type (configuration) will be built in the build tree. Possible values are empty, Debug, Release, RelWithDebInfo and MinSizeRel. Each one of them will configure the build process in a specific way, as defined in the cmake/WSNETCompilerSettings.cmake file.

In all the examples above we left this value empty, thus CMake will produce standard build type, which is the same as Debug.

The default mode is Debug for obvious reasons. However, if you wish to run intensive simulations, it is better to recompile with Release option. This option allows the compiler to optimize the executable. In order to do so, you will need to re-run cmake with the following flag:

$ cmake -DCMAKE\_BUILD\_TYPE=Release ..

The other options are:

* MinSizeRel – it will tell the compiler to optimize the size of the binaries.
* RelWithDebInfo – Release option with debug information.

As a rule, after changing your build type you will have to recompile your code:

**$** make install

Do it for both the core and also the external models you will use.

### Make options

We can compile only portions of the kernel, speeding up the compilation process for changes on just specific parts that you know it has been changed. For instance, on the kernel, one could compile only the mac models by using:

**$** make mac

The install option is also available as:

**$** make mac-install

In order to verify all available options, inside your build folder, type:

**$** make help

This will list all available make options that can be used. However, we recommend compiling the whole code to avoid leaving missing parts not compiled.

### Build folder

We use a dedicated build folder to avoid mixing temporary build files with our code base. The build folder files are generated automatically by the CMake, the Make, the compiler, etc. They are normally temporary and other auxiliary files created to help the compilation process. Therefore, there is no need to worry about them. Below you can find a figure depicting a snapshot of a build folder after the compilation:

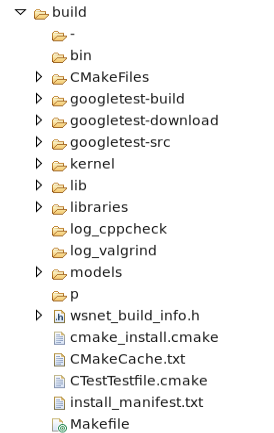


Figure 56: Build folder example

You can see the generation of different auxiliary files and folders dedicated to googletest framework, to the kernel, the models, etc. They are further used during the compilation process to generate the binaries and shared libraries.