Castalia

A simulator for Wireless Sensor Networks
and Body Area Networks

Version 3.0

User's Manual

Athanassios Boulis
August 2010
NICTA

Contents

1 Introduction

Castalia is a simulator for Wireless Sensor Networks (WSN), Body Area Networks (BAN) and generally networks of low-power embedded devices. It is based on the OMNeT++ platform and can be used by researchers and developers who want to test their distributed algorithms and/or protocols in realistic wireless channel and radio models, with a realistic node behaviour especially relating to access of the radio. Castalia can also be used to evaluate different platform characteristics for specific applications, since it is highly parametric, and can simulate a wide range of platforms. The main features of Castalia are:

- Advanced **channel model** based on empirically measured data.
    - Model defines a map of path loss, not simply connections between nodes
    - Complex model for temporal variation of path loss
    - Fully supports mobility of the nodes
    - Interference is handled as received signal strength, not as separate feature
- Advanced **radio model** based on real radios for low-power communication.
    - Probability of reception based on SINR, packet size, modulation type. PSK FSK supported, custom modulation allowed by defining SNR-BER curve.
    - Multiple TX power levels with individual node variations allowed
    - States with different power consumption and delays switching between them
    - Realistic modelling of RSSI and carrier sensing
- Extended **sensing** modelling provisions
    - Highly flexible physical process model.
    - Sensing device noise, bias, and power consumption.
- Node **clock drift**
- MAC and routing protocols available.
- Designed for **adaptation** and **expansion**.

Concerning the last bullet, Castalia was designed right from the beginning so that the users can easily implement/import their algorithms and protocols into Castalia while making use of the features the simulator is providing. Proper modularization and a configurable, automated build procedure help towards this end. The modularity, reliability, and speed of Castalia is partly enabled by OMNeT++, an excellent framework to build event-driven simulators [OMNeT++ link].

What Castalia is not: Castalia is not sensor-platform specific. Castalia is meant to provide a generic reliable and realistic framework for the first order validation of an algorithm before moving to implementation on a specific sensor platform. Castalia is not useful if one would like to test code compiled for a specific sensor node platform. For such usage there are other simulators/emulators available (e.g., Avrora).

1.1 Why a new simulator?

There is a variety of simulators that WSN researchers are using to cover their needs. Simulators that emulate a common processor found in sensor nodes (to test actual binary code written for certain platforms), simulators written in C++ or Matlab to test some first order property of an algorithm, or simulators used in traditional data networks modified in some way to serve the WSN community. So why try to build a new one?

It all started from our own needs in a WSN project. We wanted to test some communication patterns in simulation before moving in real systems. In order to do so we wanted accurate enough radio/channel models so that the simulation results would become meaningful and guide us in our search. We knew of the work by Zuniga et al. [1] that explained empirically measured data from WSN platforms (more specifically packet reception rate as a function of distance) by combining known wireless channel and radio models. We found that all the available WSN simulators were falling short of the current state of the art modelling done in sensor networks. Especially in communication where the impact to the result can be significant [5], models remain simplistic or unsuitable for short range low power communications despite the existence of proper models developed the last couple of years. This was the major reason we decided to build our own simulator. We used OMNeT++ as the base to build a reliable and fast event-driven simulator. Using OMNeT++ meant that we could just focus on the models and overall design and not on the event-driven simulation engine. Shortly after, we have decided to "up the ante", capture realistic node behaviour beyond the channel and build an open expandable and reliable simulator that has a chance of becoming a de facto standard for certain WSN simulation needs. More specifically, the need for early stage, platform-independent, algorithm/protocol validation. Since its initial inception and creation, Castalia has moved to new territories: BAN is another exciting area where realistic and reliable network-level simulators are needed. NICTA has a large scale project in BAN, participating at the same time in the IEEE standards BAN task group. With our expertise in physical layer design, measurements and modeling, we have set out to make Castalia the most realistic BAN network simulator, by modelling the temporal variations and average path losses based on real on-body measurements.

2 Overview

Castalia is using OMNeT++ as its base so it is suggested that you have a fair understanding of the basic concepts of OMNeT although this is not required, especially if you want to use Castalia in a basic way (i.e., without building your own protocols/applications)

OMNeT's basic concepts are modules and messages. A simple module is the basic unit of execution. It accepts messages from other modules or itself, and according to the message, it executes a piece of code. The code can keep state that is altered when messages are received and can send (or schedule) new messages. There are also composite modules. A composite module is just a construction of simple and/or other composite modules.

2.1 Structure

Castalia's basic module structure is shown in the diagram below.

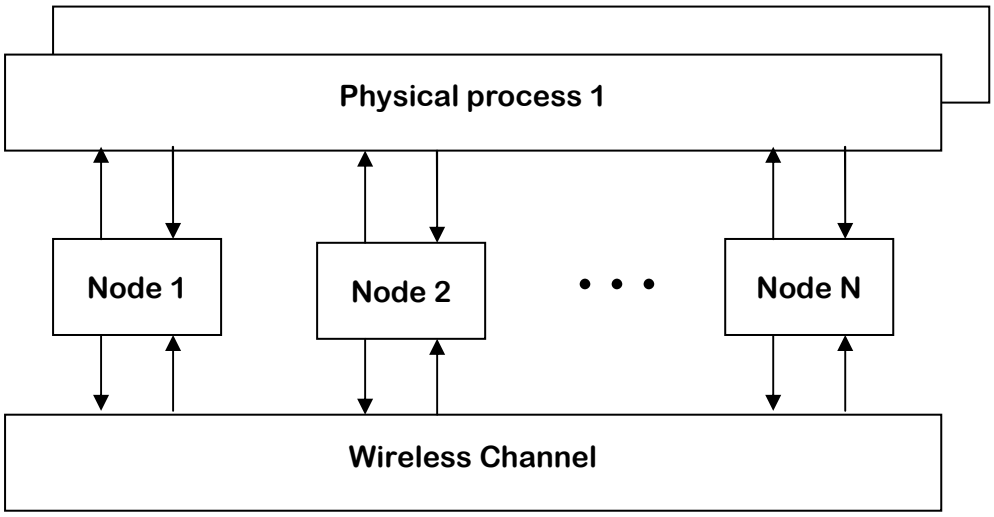

Figure 1: The modules and their connections in Castalia

Notice that the nodes do not connect to each other directly but through the wireless channel module(s). The arrows signify message passing from one module to another. When a node has a packet to send this goes to the wireless channel which then decides which nodes should receive the packet. The nodes are also linked through the physical processes that they monitor. For every physical process there is one module which holds the "truth" on the quantity the physical process is representing. The nodes sample the physical process in space and time (by sending a message to the corresponding module) to get their sensor readings.

There can be multiple physical processes, representing the multiple sensing devices (multiple sensing modalities) that a node has.

The node module is a composite one. Figure 2 shows the internal structure of the node composite module. The solid arrows signify message passing and the dashed arrows signify simple function calling. For instance, most of the modules call a function of the resource manager to signal that energy has been consumed. The Application module is the one that the user will most commonly change, usually by creating a new module to implement a. new algorithm. The communications MAC and Routing modules, as well as the Mobility Manager module, are also good candidates for change by the user, again usually by creating a new module to implement a new protocol or mobility pattern. Castalia offers support for building your own protocols, or applications by defining appropriate abstract classes (more on this topic in Chapter 5). All existing modules are highly tuneable by many parameters.

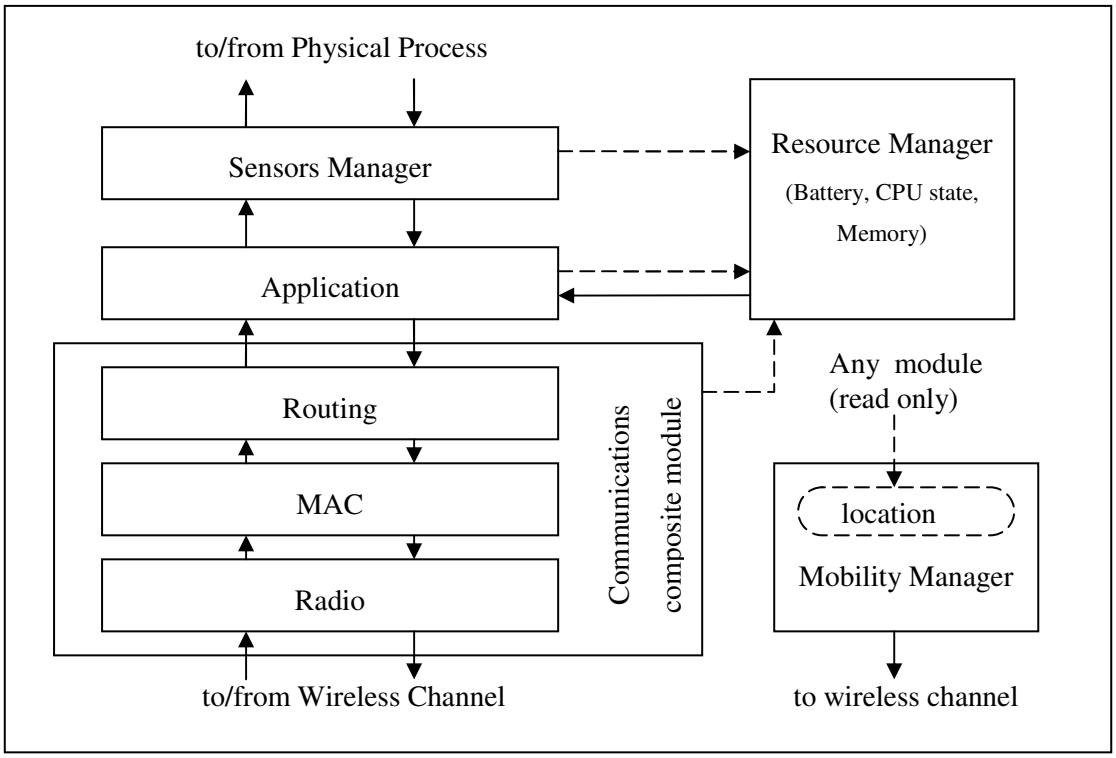

Figure 2: The node composite module

This structure depicted in the figures above and described in this chapter is implemented in Castalia with the use of the OMNeT++ NED language. With this language we can easily define modules, i.e., define a module name, module parameters, and module interface (gates in and gates out) and possible submodule structure (if this is a composite module). Files with the suffix "`.ned`" contain NED language code. The Castalia structure is also reflected in the hierarchy of directories in the source code. Every module corresponds to a directory which always contains a `.ned` file that defines the module. If the module is

composite then there are subdirectories to define the submodules. If it is a simple module then there is C++ code (.cc, .h files) to define its behaviour. This complete hierarchy of .ned files defines the overall structure of the Castalia simulator. Normally the user will not alter these files. Nevertheless, these files are dynamically loaded and processed (using a feature of OMNeT) so that any change does not require the recompilation of Castalia (unless of course new simple modules with new functionality appear).

In the rest of this document we will use these formatting conventions:

Commands given at the shell and output produced by them:

```
current_path$ commandline
Outputlines
Outputlines
```

Portions of configuration files (usually named omnetpp.ini):

```
Parameter1 = 1
Parameter2 = 2
```

C++ code or NED scripting code:

```
Code
```

Paths, file names, and parameter names will be written in `Courier` fonts. `Castalia/` is assumed to be the top-most level directory where all the Castalia files are installed.

3 Using Castalia

We assume here that you successfully installed both OMNeT and Castalia. Refer to the Installation documents for trouble shooting. Since the release of Castalia 3.0, handling of input and output has changed considerably. We advise you to read this section even if you had previous experiences with Castalia 2.3b or earlier releases. Since Castalia 3.0, we have two scripts to help with running simulations and interpreting the results. They are called `Castalia` and `CastaliaResults` respectively. They both reside in `Castalia/bin/`[1]. The executable of the simulator is called `CastaliaBin` residing in `Castalia/` but you will not need to invoke it directly.

3.1 Running the first simulation

It's time to dive in and run our first simulation. Go to the directory `Castalia/Simulations/radioTest`. It should include one file: omnetpp.ini. This is a configuration file that defines our simulation scenario(s). Let us run the input script with no arguments and see what we get:

```
~/Castalia/Simulations/radioTest$ ../../bin/Castalia
List of available input files and configurations:
* omnetpp.ini
        General
        InterferenceTest1
        InterferenceTest2
        CSinterruptTest
        varyInterferenceModel
```

Executed with no arguments, the script searches the current directory for valid configuration files. If it finds a file, it then parses it and prints the name of the configurations contained in it (more about configurations shortly). In our case it found just one file with five configurations.

---

[1] It is suggested that you add Castalia/bin/ in your PATH env variable. In the command line examples we give in this manual, we assume that Castalia/bin is in the path. If you are using bash you simply do

`$ export PATH=$PATH:~/Castalia/bin`

(assuming Castalia is installed under your home dir)

Also do not forget to add this export statement in your `.bash_profile` file

To see how we can use the Castalia input script to do something more exciting, ask for help. Run it with –h as argument.

```
~/Castalia/Simulations/radioTest$ ../../bin/Castalia -h
Usage: Castalia [options]

Options:
  -h, --help            show this help message and exit
  -c CONFIG, --config=CONFIG
                        A list of configuration names to use, comma
                        Separated configurations will be joined
                        together, configurations listed in brackets
                        will be interleaved
  -i FILE, --input=FILE
                        Select input configuration file, default is
                        omnetpp.ini
  -d, --debug           Debug mode, will display results from each
                        CastaliaBin execution
  -r N, --repeat=N      Number of repetitions for each unique
                        scenario
```

We see that we can instruct the Castalia script to use a specific file with the –i switch (although we do not need in our case, omnettpp.ini is the default) and use the –c switch to choose a configuration within this file. For the moment do not mind the help text mentioning that a list of configuration can be given, and just run it with one configuration. Let's use `General`, which is always present in an omnetpp.ini file and is considered the default configuration.

```
~/Castalia/Simulations/radioTest$ ../../bin/Castalia -c General
Running configuration 1/1
```

You've just run your first simulation! Let's see what's new in our directory.

```
~/Castalia/Simulations/radioTest$ ls
100806-222319.txt  Castalia-Trace.txt  omnetpp.ini
```

There are 2 new files created: `100806-222319.txt` is Castalia's standard output file and its name is in the form `YYMMDD-HHMMSS.txt`. You can freely rename this file if you wish. You can also open it to read it (it is human readable) but it is not supposed to be viewed this way. Normally you will process this file with `CastaliaResults`. We will see how to do this in section 3.3. The other file produced is a trace file named `Castalia-Trace.txt`. It contains a trace of all events that you requested to be recorded by "turning on" some

parameters in the omnetpp.ini file. By default all tracing is turned off, but for this simulation example we wanted to activate just the tracing from the application module of node 0. Open the file and have a look.

```
~/Castalia/Simulations/radioTest$ less Castalia-Trace.txt

0.027540267327  SN.node[0].Application       Not sending packets
3.868527053713  SN.node[0].Application       Received packet from node 1, SN = 18
4.068529304763  SN.node[0].Application       Received packet from node 1, SN = 19
4.268531555813  SN.node[0].Application       Received packet from node 1, SN = 20
4.468533806863  SN.node[0].Application       Received packet from node 1, SN = 21
4.668536057913  SN.node[0].Application       Received packet from node 1, SN = 22
…
```

Each line is a trace event. The first item in the line is the simulation time that the event happened. Second item is the full name of the module that produced this trace line. In the example above all lines are produced by the Application module of node 0. finally the last item is the trace message itself. In the example above most messages notify us of packet received by node 1, also printing the packet's serial number.

But what is this simulation supposed to do?

We created the ratioTest simulation scenarios so users can see the results of realistic modeling and also see some of Castalia's features in action. The first scenario (the General configuration we just run) tests reception as a receiver (node 0) moves through the area of two transmitters (nodes 1 and 2). The transmitters are placed far enough so there is no interference between them. The receiver moves in a straight line back and forth and when it is close to each of the two transmitters it should receive their packets. For this simulation we have turned off shadowing effects so that the successful-reception-distance is identical for the two transmitters (still there are random effect in the radio reception, so received packets by the two transmitters are not identical). The InterferenceTest1 and InterferenceTest2 scenarios have the receiver static and one of the transmitters moving. The idea there is to show the effect of interference with the use of mobility. In InterferenceTest1 the interfering node passes through the middle of the distance between node 1 and 0, and thus just causes collisions. In InterferenceTest2 though it passes very close to the receiver (compared to the other transmitter) and thus even though initially it creates collisions, when sufficiently close, its SINR is high enough for its packets to be received (despite interference from the transmissions of the static node 1). Figure 3 illustrates these interactions.

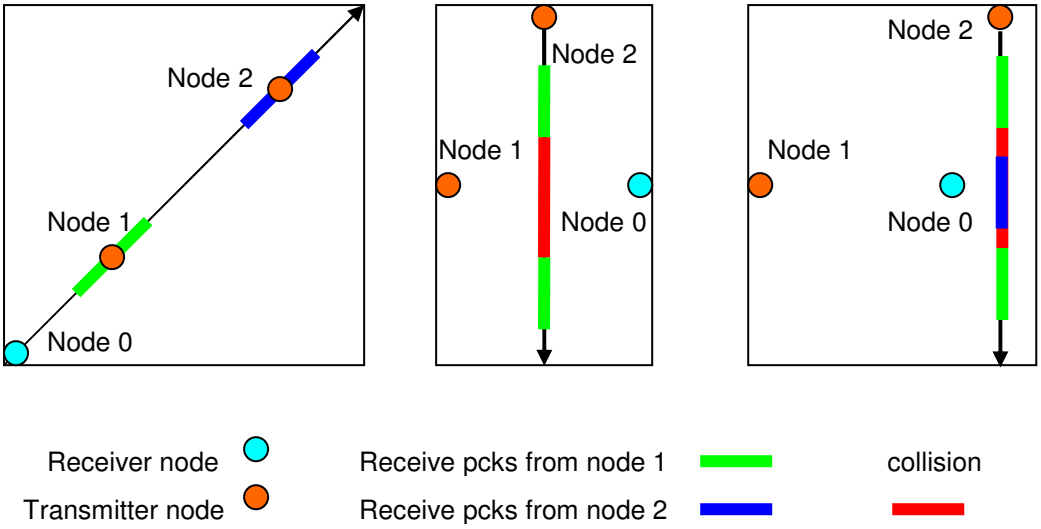

Figure 3: The General, InterferenceTest1, and InterferenceTest2 configurations

Run the other two simulation scenarios and look at the resulting Castalia-Trace.txt files (delete the old trace files if you do not want the new traces appended in the old file).

```
~/Castalia/Simulations/radioTest$ rm Castalia-Trace.txt
~/Castalia/Simulations/radioTest$ Castalia -c InterferenceTest1
Running configuration 1/1
```

Are the traces as expected? You should be able to see the patterns of reception and breaking of reception as depicted in figure 3.

Let's us look more deeply into the omnetpp.ini file.

3.2 Understanding the configuration file

As described in Chapter 2, Castalia has a modular structure with many interconnecting modules. Each one of the modules has one or more parameters that affect its behavior. An NED file defines the basic structure of a module by defining its input/output gates and its *parameters*. At the NED file we can also define *default values* for the parameters. A user can redefine parameter values in a configuration file. This way we can build a great variety of simulation scenarios. Open the file `Simulations/radioTest/omnetpp.ini`

Note that everything after a # character is a comment. The configuration file starts with the General section. Every OMNeT configuration file has a General section. There you define the basic scenario that this file describes. Parameters such as the simulation time and number of nodes do not have any default values and must be defined in this section. Below are the first few lines from the `radioTest/omnetpp.ini` file (comments removed)

```
[General]
include ../Parameters/Castalia.ini

sim-time-limit = 100s

SN.field_x = 200        # meters
SN.field_y = 200        # meters

SN.numNodes = 3
```

The first line in the General section is an include command. As the name suggests, the command includes the contents of the file it gets as an argument. In the particular example we include a file we named Castalia.ini and it contains some basic parameter assignments[2] that every Castalia simulation needs. So when you create a Castalia configuration file, always include Castalia.ini in it.

Then we define the simulation time (here assigned to 100 secs). This is the only generic OMNeT parameter we assign in a Castalia configuration file. We can see that all other parameter start their name with SN. SN is the topmost composite module (or "network" as OMNeT calls it). The name SN stands for Sensor Network. There are a few parameters that the network takes such as the field size, the number of nodes, and the deployment type. The field size is given by 3 real numbers, each for axes x,y,z. One can define only 2 of the axes if

[2] Castalia.ini assigns some parameters affecting general OMNeT execution, as well as parameters that map the different Random Number Generator (RNGs) to modules.

desired (the third one will take the value 0 by default). Number of nodes is given by an integer. The parameter `SN.deployment` is a string that describes where the nodes are placed on the field. If left undefined (as in our example above) then the location of the nodes is determined by the xCoor, yCoor, zCoor parameters of each node. These are usually manually assigned in the omnetpp.ini file. When defined, `SN.deployment` can be ones of the following types:

"uniform" → nodes are placed in the field using a random uniform distribution

"NxM"→ N and M are integer numbers. Nodes placed in a grid of N nodes by M nodes

"NxMxK" → same as above but for 3 dimensions

"randomizedNxM" → a grid deployment with randomness in the position of the nodes. If Gx and Gy is the grid spacing in axes x and y, and (Xi, Yi) are the grid position of node i then the randomized grid position is (Xi+Rx, Yi+Ry) where Rx and Ry are numbers uniformly drawn from [(-1/3)Gx…(1/3)Gx] and [(-1/3)Gy…(1/3)Gy]

"randomizedNxMxK" → same as above but for 3 dimensions

"center" → node put in the centre of the field

We can even define *mixed* deployments by using this string format:

[N1..N2]->type;[N2..N3]->type;… where Nx are node IDs, and types is one of the options given above

After we set these top level parameters, we go into setting module parameters. When setting a module parameter, the full name of the module is given revealing the module hierarchy structure. Here is some example from our configuration file, setting some parameters of the wireless channel.

```
# important wireless channel switch to allow mobility
SN.wirelessChannel.onlyStaticNodes = false
SN.wirelessChannel.sigma = 0
SN.wirelessChannel.bidirectionalSigma = 0
```

We will explain the meaning and use of all parameters in Chapter 4 (Modeling in Castalia). In this section we present some basic concepts and syntax rules of Castalia configuration files.

For instance, how do we set radio parameters? Remember that the radio is part of the Communication composite module of *every* node. So there are as many Radio modules as there are nodes. How do we set all of them easily? Here's how:

```
SN.node[*].Communication.Radio.RadioParametersFile =
                                    "../Parameters/Radio/CC2420.txt"
SN.node[*].Communication.Radio.TxOutputPower = "-5dBm"
```

These two lines set 2 parameters of the Radio module. The RadioParametersFile is a specially formatted file defining the basic operational properties of a radio (more on Chapter 4). The second parameter, TxOutputPower is sets the power that the radio transmits its packets. Notice how we access all the radio modules in all the nodes with the use of [*].

The sensor network compound module (SN) contains many Node sub-modules. These are addressed in the form of an array. For instance, here's how to set the parameter `xCoor` (the x coordinate of the location of a node) of the node with ID=9:

```
SN.node[9].xCoor = 10.5
```

However, most of the time we need to massively assign values for all the nodes of the sensor network. This is possible by using wildcards like:

`[*]` → all indexes

`[3..5]` → indexes 3,4,5

`[..4]` → indexes 0, 1, 2, 3, 4

`[5..]` → indexes 5 till the last one

Reading on in our example omnetpp.ini file we see that parameters relating to the maximum allowed packet size are set in all communication layers:

```
SN.node[*].Communication.Routing.maxNetFrameSize = 2500
SN.node[*].Communication.MAC.maxMACFrameSize = 2500
SN.node[*].Communication.Radio.maxPhyFrameSize = 2500
```

We have already set the kind of radio we want to use, by setting RadioParametersFile. How do we instruct Castalia which MAC protocol and which Routing protocol to use? This is achieved by two Communication module parameters

`SN.node[*].Communication.MACProtocolName`

`SN.node[*].Communication.RoutingProtocolName`

These are set by default to "`BypassMAC`" and "`BypassRouting`". These are module names that in essence implement no MAC functionality and no Routing functionality. For the radioTest simulation we did not need any MAC or Routing so we left these parameters assigned to their default value (i.e., they are not mentioned in the omnetpp.ini file). If you want to see the default values for any parameter just go to the .ned file that describes that module. For the parameters mentioned above you can look in:

`src/node/communication/CommunicationModule.ned`

If you look at the BANtest/omnetpp.ini you will see MACProtocolName being assigned

```
SN.node[*].Communication.MACProtocolName = "BaselineBANMac"
```

The important thing to note here is that by assigning a name to this parameter we are also choosing a specific module for our MAC. So by altering these parameters we can dynamically control the composition of modules in our simulation[3]. The consequence of this feature is that depending on the module you choose, the parameters available to you are also changing since different MAC modules support different parameters. There are four kinds of modules that are dynamically selected in a Castalia configuration file. We have already seen the 2 kinds: MAC and Routing modules. The other 2 are Application modules and MobilityManager modules.

Here is how we set our radioTest simulation to use the Application module we want:

```
SN.node[*].ApplicationName = "ThroughputTest"
SN.node[*].Application.packet_rate = 5
SN.node[*].Application.constantDataPayload = 2000
# application's trace info for node 0 (receiving node)
# is turned on, to show some interesting patterns
SN.node[0].Application.collectTraceInfo = true
```

By setting the ApplicationName we also choose the application module. ThroughputTest application has all nodes sending traffic sending traffic to a sink node. We can easily modify the packet rate and the packet size as you can notice. The sink node is by default node 0 and we have not changed this here. Also notice how the last line shown above, activates the collecting of traces for the Application module of node 0. This is why the Castlia-Trace.txt files we examined in section 3.1 were created and this is why they only contained events from the application module of node 0. *All Castalia modules* have a `collectTraceInfo` parameter. By default, collection of traces is deactivated (parameter set to false).

Then follows a section where we set the starting locations for our 3 nodes manually (remember that we set `SN.deployment` to "custom"). We write "starting locations" because the nodes can be mobile. Indeed, we see that node 0 is using `LineMobilityManager` as the mobility manager module. The other two nodes have the default mobility manager: `NoMobilityManager`, which is used to describe static nodes.

[3] This is achieved thanks to OMNeT's dynamic module loading capability and also its ability to define parametric module names in .ned files.

```
SN.node[0].xCoor = 0
SN.node[0].yCoor = 0
SN.node[1].xCoor = 50
SN.node[1].yCoor = 50
SN.node[2].xCoor = 150
SN.node[2].yCoor = 150

SN.node[0].MobilityManagerName = "LineMobilityManager"

SN.node[0].MobilityManager.updateInterval = 100
SN.node[0].MobilityManager.xCoorDestination = 200
SN.node[0].MobilityManager.yCoorDestination = 200
SN.node[0].MobilityManager.speed = 15
```

The `LineMobilityManager` module moves the node in a straight line segment (back and forth). The straight line segment is defined by the starting location of a node and the destination location given by the xCoorDestination, yCoorDestination parameters of the mobility module. We also define the speed and the update interval that the wireless channel module will be updated of the node's current position.

After these lines we see a new section: a new *configuration*.

```
[Config InterferenceTest1]
SN.node[0].MobilityManagerName = "NoMobilityManager"
SN.node[1].MobilityManagerName = "NoMobilityManager"
SN.node[2].MobilityManagerName = "LineMobilityManager"

SN.node[0].xCoor = 10
SN.node[0].yCoor = 50

SN.node[1].xCoor = 0
SN.node[1].yCoor = 50

SN.node[2].xCoor = 5
SN.node[2].yCoor = 0

SN.node[2].MobilityManager.updateInterval = 100
SN.node[2].MobilityManager.xCoorDestination = 5
SN.node[2].MobilityManager.yCoorDestination = 100
SN.node[2].MobilityManager.speed = 5
```

In this configuration, node 2 is the mobile one, instead of node 0, and nodes are much closer to each other (so that we can simulate interesting interference behaviour).

Yet another configuration section follows with slightly different node locations, which we do not show here. Instead we draw you attention at the end of the file, where a very short configuration section resides. Here it is:

```
[Config varyInterferenceModel]
SN.node[*].Communication.Radio.collisionModel = ${InterfModel=0,1,2}
```

Notice how we do not assign just one value to the parameter but a series of values. This is a feature of OMNeT 4.0 and later versions. You can look at the OMNeT manual for a complete list of the ways you can assign parameters, but the essence of this multiple-value assignment is that simulation will run many times, each time assigning one of the values. So given the example above we will run the simulation 3 times. The name "InterfModel" in the curly braces will just be used as a label in the output produced and helps us parse the output more easily.

If you have more than one parameter that takes multiple values, then all combinations will be run. In the example below we have 2 parameters taking 3 values each, which means 9 possible parameter combinations, thus 9 simulation runs.

```
SN.node[1].Communication.Radio.TxOutputPower = ${TxPower="-5dBm", "-10dBm", "-15dBm"}
SN.node[0].Communication.Radio.CCAthreshold = ${CCAthreshold=-95, -90, -85}
```

You can even do more complicated things such as putting constraints, so that not all possible combinations are executed. Here's an example from `BANtest/omnetpp.ini`:

```
SN.node[*].Communication.MAC.scheduledAccessLength = ${schedSlots=6,5,4,3}
SN.node[*].Communication.MAC.RAP1Length = ${RAPslots=2,7,12,17}
constraint = $schedSlots * 5 + $RAPslots == 32
```

The example above will only execute 4 combinations: (6,2), (5,7), (4,12), (3,17).

3.3 Using the Castalia and CastaliaResults scripts

OMNeT allows you to choose just one of the configurations defined in a configuration file. That is, if you use the simulator binary directly (i.e. CastaliaBin) you can choose an input configuration file and one configuration. When choosing a configuration the parameters from the General section are used and set, and if the configuration redefines them, then their value is overwritten. Although this feature is an improvement over older versions, its usefulness is limited. Notice for example, the configuration on varying the collision model we examined earlier. If we choose it using OMNeT's standard feature it will only be applied in the General section scenario. If we would like the same effect in the Interference scenarios then we would have to explicitly include that line in each configuration. If we also wanted to vary the Tx power but not vary the collision model, then we would have to explicitly write another set of configurations. The problem is that you cannot easily *combine* configurations. You have to explicitly write the combinations. Furthermore, you cannot easily execute more than one configuration at a time. We can overcome these restrictions by using the Castalia script.

With the Castalia script we can easily give a list of configurations to be combined.

For example, we can write "Castalia –c InterferenceTest1,varyInterferenceModel" or "Castalia –c InterferenceTest2,varyInterferenceModel" no need to create the combination explicitly in the omnetpp.ini file. Moreover we have the ability to define more than one configurations to be executed by putting them in braces. Castalia –c [A,B] will execute the simulation with configuration A and then another simulation with configuration B.

Castalia –c [A,B]C,D[E,F] will execute 4 combined configurations:

A,C,D,E ← this is a combination (or concatenation) of the 4 simple configs listed

B,C,D,E

A,C,D,F

B,C,D,F

If you try to combine two or more configurations that have at least one overlapping parameter (i.e. same parameter is defined in two or more configurations) then you will get an error message from the script. For example if you tried:

```
radioTest$ Castalia -c InterferenceTest1,InterferenceTest2
ERROR: conflicting values for parameter 'SN.node[2].xCoor' in base
congigurations: '5' and '22'
```

Let us use the power of the Castalia script to run a useful simulation:

```
$ Castalia -c [InterferenceTest1,InterferenceTest2]varyInterferenceModel
Running configuration 1/2
Running configuration 2/2
```

A new output file was created (in our system100809-004640.txt) and Castalia-Trace.txt got appended with new traces. How would you check the results? You could open the output file or the trace, but there is a lot of text to parse and understand. Remember that for each of the configurations we have 3 collision models tried, so in total we had 6 simulations run. We need a way to process and summarize the results. This is what the script CastaliaResults is for. Let's run it with no arguments to see what it gives us:

```
~/Castalia/Simulations/radioTest$ CastaliaResults

Castalia output files in current directory:
+------------------+--------------------------------+------------------+
|                  | Configuration                  | Date             |
+------------------+--------------------------------+------------------+
| 100807-160236.txt | General                       | 2010-08-07 16:02 |
| 100808-014651.txt | InterferenceTest1             | 2010-08-08 01:46 |
| 100809-004640.txt | [InterferenceTest1,Interference | 2010-08-09 00:46 |
|                  |  Test2]varyInterferenceModel   |                  |
+------------------+--------------------------------+------------------+
```

It gives us a list of valid Castalia output files (also called result files) with information about the configuration that created them and the date they were created. You can use the –h option to get help on the arguments the script can take. Let's run the script giving it one of the results file as input (use the –i switch)

```
radioTest$ CastaliaResults -i 100809-004640.txt

+--------------------+--------------------------------+------------+
|             Module |                         Output | Dimensions |
+--------------------+--------------------------------+------------+
|        Application | Application level latency, in ms | 1x2(11)    |
|                    |                Packets received | 1x2        |
| Communication.Radio |                RX pkt breakdown | 3x1(6)     |
|                    |                       TXed pkts | 2x1        |
|    ResourceManager |                 Consumed Energy | 3x1        |
+--------------------+--------------------------------+------------+
NOTE: select from the available outputs using the -s option
```

CastaliaResults parses the file and finds out what output is recorded by the different modules. Application for example produces two kinds of output relating to packet latency and packets received. Each output has its dimensions NxM.

N is the number of modules that produced this output. Modules that are instantiated only once (e.g., the wirelessChannel) will always have N=1. Modules that are instantiated n times (e.g., the node and all of its submodules) can have N equal up to n. In the example above, even though there are 3 Application modules (recall that radioTest simulations instantiate 3 nodes), only one of them is producing output. This is the Application module of node 0, since this is the only one receiving packets.

M, is the number of different indices, an output from a *single* module has. When defining an output, it is possible to give it an index parameter. Usually this is used to differentiate output relating to different nodes. For example the "Packets received" output has an index specifying the sender of the packet. This way we can keep track of how many packets did a node receive from the different senders. If an output does not have an index (e.g., the consumed energy output of a node) then M=1.

Finally, if an output from a single module and for a single index is not scalar, then we write its multiplicity in parentheses. For example, the latency output is a histogram with 11 time buckets. As another example, the Radio received packet breakdown has 6 different types of packets. The types are naming the different conditions the packets failed or succeeded.

The note in the output above urges us to use the –s switch ('s' standing for 'show') to select among the possible outputs. You just give it a regular expression and it will present the results from the output names that match the regular expression. Let's say we want to see what the application packets results are:

```
radioTest$ CastaliaResults -i 100809-004640.txt -s packet
Application:Packets received
+------------------+--------------+---------------+---------------+
|                  | InterfModel=0 | InterfModel=1 | InterfModel=2 |
+------------------+--------------+---------------+---------------+
| InterferenceTest1 | 335.5        | 12            | 99.5          |
| InterferenceTest2 | 332.5        | 11.5          | 24.5          |
+------------------+--------------+---------------+---------------+
```

We see the results for both interference tests for all 3 collision modes in a nice 2×3 matrix. The results in each cell are averages for all modules and all indices. In our case it is the average received packets at node 0, from both nodes 1 and 2. This table gives us an understanding of the situation but it would be more informative if we could see the results for

the individual sender nodes and not the average. We can do this by using the –n switch (‘n’ standing for ‘node’). Using this switch means that the dimensions of the output are expanded.

Let’s try it:

```
radioTest$ CastaliaResults -i 100809-004640.txt -s packets -n
Application:Packets received
+---------------------------------+---------+---------+
|                                 | Index=1 | Index=2 |
+---------------------------------+---------+---------+
| InterferenceTest1,InterfModel=0 | 499     | 172     |
| InterferenceTest1,InterfModel=1 | 24      | 0       |
| InterferenceTest1,InterfModel=2 | 199     | 0       |
| InterferenceTest2,InterfModel=0 | 494     | 171     |
| InterferenceTest2,InterfModel=1 | 23      | 0       |
| InterferenceTest2,InterfModel=2 | 25      | 24      |
+---------------------------------+---------+---------+
```

We see now that the table of results became 6×2 and now the columns are labeled with an index which is the ID of the sender node. We see that when collision model = 0 is used (=no collisions) then node 0 receives the most packets from both node 1 and 2. Node 2 moves, so it is often outside the range of node 0. That’s why fewer packets are received from it. There is some randomness at the Radio module (packet reception probability is based on SNR) so the packets received from node 1 are not exactly equal in both Interference Tests. When collision model = 1 is used (= simple model based on the notion of interference range) that creates more collisions than reality. We see that very few packets get through. Finally, when we use model=2 (additive interference model) then this takes into account the dynamic SINR of each packet. We see that node 2 actually manages to get some packets through as we described in Figure 3. For a detailed explanation of the different collision models refer to Chapter 4.

Let us now move to a different simulation scenario and a different application altogether. The connectivity map application: This application is designed to produce a connectivity map of the network by finding out the link quality between nodes. Each node is programmed to transmit 100 packets at a unique timeslot, so that there are no collisions with other nodes. A node –when not transmitting– constantly listens for incoming packets, and when one is received, it increases the counter of the packets heard from the sender node. At the end of the simulation the counters are written as application output using the sender node as index. Go to `Simulations/connectivityMap/` and inspect the omnetpp.ini file

The configuration file describes a grid of 3nodes×3nodes (9 nodes in total). The General configuration has a wireless channel with no shadowing randomness and the radio using a low Tx power. Let's run it and inspect the results.

```
connectivityMap$ Castalia -c General
Running configuration 1/1

connectivityMap$ ls
100809-145319.txt  omnetpp.ini

connectivityMap$ CastaliaResults -i 100809-145319.txt
+--------------------+------------------+------------+
|              Module |           Output | Dimensions |
+--------------------+------------------+------------+
|         Application | Packets received | 9x9        |
| Communication.Radio | RX pkt breakdown | 9x1(3)     |
|                     |        TXed pkts | 9x1        |
|     ResourceManager |  Consumed Energy | 9x1        |
+--------------------+------------------+------------+
NOTE: select from the available outputs using the -s option
```

We see that the Application module output Packets received is 9×9. This means that there are 9 modules reporting and for each module there are 9 indices (each corresponding to other nodes + the self node). Simply printing the result of that output will give us the average on these 9×9 (= 81) outputs. This will be the average link quality over all the links[4].

```
connectivityMap$ CastaliaResults -i 100809-145319.txt -s packets
Application:Packets received
+--------+
|        |
+--------+
| 88.225 |
+--------+
```

This in not such an interesting result, on its own. It would be much more interesting if we could see the quality of individual links. How many packets out of the 100 did each link (node A→ node B) was able to receive. We need to show individual outputs, not the average.

[4] It is the average of link quality for links that are not 0%, i.e. at least one packet was successfully received.

We already mentioned that if we want to expand an output to each dimensions we use the –n switch. Let's try it.

```
connectivityMap$ CastaliaResults -i 100809-145319.txt -s packets -n

Application:Packets received
+--------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
|        |Node=0 |Node=1 |Node=2 |Node=3 |Node=4 |Node=5 |Node=6 |Node=7 |Node=8 |
+--------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
|index=0 | 0     | 100   | 0     | 99    | 73    | 0     | 0     | 0     | 0     |
|index=1 | 100   | 0     | 100   | 70    | 100   | 72    | 0     | 0     | 0     |
|index=2 | 0     | 100   | 0     | 0     | 73    | 100   | 0     | 0     | 0     |
|index=3 | 100   | 75    | 0     | 0     | 100   | 0     | 100   | 70    | 0     |
|index=4 | 78    | 100   | 72    | 100   | 0     | 100   | 66    | 100   | 64    |
|index=5 | 0     | 64    | 100   | 0     | 100   | 0     | 0     | 65    | 100   |
|index=6 | 0     | 0     | 0     | 100   | 72    | 0     | 0     | 100   | 0     |
|index=7 | 0     | 0     | 0     | 69    | 100   | 72    | 100   | 0     | 100   |
|index=8 | 0     | 0     | 0     | 0     | 75    | 100   | 0     | 100   | 0     |
+--------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
```

Now we can see a map of connectivity! Each column is the results reported by a node. That is, each column i, is the link qualities of all incoming links to node i. Looking at the first column we can see that node 0 can hear nodes 1 and 3 perfectly and node 4 at 78%. This is expected as nodes 1 and 3 are the closest to node 0 (being immediately to the right and down of node 0) while node 4 is the next closest node (being on the right-down diagonal). Node 4 has the best connectivity (as expected) as it can listen to all 8 nodes around it.

What would happen if we varied the Tx Power? Or varied the sigma of the wireless channel (i.e. the randomness of the shadowing)? If you look into the omnetpp.ini file you will see that we have appropriate configurations for these questions. So we can run:

```
connectivityMap$ Castalia -c varyTxPower
Running configuration 1/1
connectivityMap$ Castalia -c varySigma
Running configuration 1/1
```

Using Castaliaresults, we can look into the application received packets produced by the first simulation. This will result in a long table since the 9x9 results will be multiplied 4 time (each time for a different TxOutputPower value) If we wish to limit the portion of the table shown, we can filter the rows with the –f switch. For instance, using –f 5dBm will print only rows that contain "5dBm" in their labels:

```
connectivityMap$ CastaliaResults -i 100810-020129.txt -s packet -n -f 5dBm
```

3.4 Advanced usage

Let us move to a different simulation scenario. Go to `Castalia/Simulations/valuePropagation/` and examine the omnetpp.ini file there. It defines a grid of 4 nodes by 4 nodes, uses a MAC called "TunableMAC" and an application called "valuePropagation". The application works in the following way: All nodes sample their temperature sensors periodically. If a sensed value is above the threshold of 15ºC then this value needs to be broadcasted. If a node receives this value from any other node it tries to broadcast it and then sets a flag that it has done its duty. Depending on the behaviour of the MAC and the condition of the channel between the nodes, the value is propagated in the network. The interesting thing to explore here is how the value propagation is affected by MAC parameters and how much energy is consumed each time. TunableMAC implements a generic duty cycle MAC, and as the name suggests, it has many parameters that the user can tune. With the current settings of the omnetpp.ini file, only node 0 is sampling a value beyond 15 (node 0 samples a value of 40 + noise), the rest are sampling a value of 0 + noise. So node 0 is the only source of the special value to be propagated. Notice that the omnetpp.ini file provides 3 configurations:

```
[Config varyDutyCycle]
SN.node[*].Communication.MAC.dutyCycle = ${dutyCycle= 0.02, 0.05, 0.1}


[Config varyBeacon]
SN.node[*].Communication.MAC.beaconIntervalFraction = ${beaconFraction= 0.2,
0.5, 0.8}


[Config varyTxPower]
SN.node[*].Communication.Radio.TxOutputPower = ${TXpower="-1dBm","-5dBm"}
```

Each of the configuration varies a different parameter, so we can combine them all together with the `Castalia` script and have one big simulation that runs all 3×3×2 = 18 parameter possibilities:

```
valuePropagation$ Castalia -c varyDutyCycle,varyBeacon,varyTxPower
Running configuration 1/1
```

Let us check the resulting output file:

```
/valuePropagation$ CastaliaResults -i 100812-102156.txt
```

```
+---------------------+------------------+------------+
|              Module |           Output | Dimensions |
+---------------------+------------------+------------+
|         Application |        got value | 16x1       |
| Communication.Radio | RX pkt breakdown | 16x1(6)    |
|                     |        TXed pkts | 16x1       |
|     ResourceManager |  Consumed Energy | 16x1       |
+---------------------+------------------+------------+
NOTE: select from the available outputs using the -s option
```

We see that the application module reports an output called “got value”. We also see that all of the 16 application modules (one for each node) have reported this output. This is a simple 1/0 output, reporting 1 if a node has a value above the threshold (either by sensing, or received by another node) and 0 if it is below the threshold. Looking at an average of this value across all sensor nodes gives you the fraction of the nodes that have received the value. Thus this is a metric of value propagation. Let’s see what are the results:

```
/valuePropagation$ CastaliaResults -i 100812-102156.txt -s got

Application:got value
+----------------------------------+-----------------+-----------------+
|                                  | TXpower="-5dBm" | TXpower="-1dBm" |
+----------------------------------+-----------------+-----------------+
| beaconFraction=0.2,dutyCycle=0.02 | 0.188           | 0.25            |
| beaconFraction=0.2,dutyCycle=0.05 | 0.125           | 0.063           |
| beaconFraction=0.2,dutyCycle=0.1  | 0.063           | 0.75            |
| beaconFraction=0.5,dutyCycle=0.02 | 0.375           | 0.938           |
| beaconFraction=0.5,dutyCycle=0.05 | 0.063           | 0.313           |
| beaconFraction=0.5,dutyCycle=0.1  | 0.188           | 0.938           |
| beaconFraction=0.8,dutyCycle=0.02 | 0.813           | 1               |
| beaconFraction=0.8,dutyCycle=0.05 | 0.75            | 0.063           |
| beaconFraction=0.8,dutyCycle=0.1  | 0.688           | 1               |
+----------------------------------+-----------------+-----------------+
```

The table shows the propagation metrics for all 18 different parameter scenarios. For example we see that when beaconFraction=0.2,dutyCycle=0.02, TXpower="-1dBm", 0.25 of the nodes (4/16nodes) got the value. For a couple of cases with high beaconFraction value we see that all of the nodes got the value.

3.4.1 Manipulating the display of a results table

It is interesting to note here that there are 3 parameter dimensions in the results (beaconFraction, dutyCycle, TXpower) yet we can only show 2D tables. In the example above `CastaliaResults` has chosen to put TXPower in the columns and put beaconFraction in the "outer loop" of the rows enumeration. What if we wanted some other order and display of the results. The user has full control with the --order switch. This switch is followed by a comma-separated list of the label. The first label will be the columns of the table, the second label will be the outer loop in the rows enumeration of the table and so on. You do not have to define all labels. If you define just one, this will be the columns and the rest will be ordered in a default way. Let's try this option:

```
/valuePropagation$ CastaliaResults -i 100812-102156.txt -s got --order=duty,TX

Application:got value
+----------------------------------+----------------+----------------+----------------+
|                                  | dutyCycle=0.02 | dutyCycle=0.05 | dutyCycle=0.1 |
+----------------------------------+----------------+----------------+----------------+
|TXpower="-5dBm",beaconFraction=0.2| 0.188          | 0.125          | 0.063         |
|TXpower="-5dBm",beaconFraction=0.5| 0.375          | 0.063          | 0.188         |
|TXpower="-5dBm",beaconFraction=0.8| 0.813          | 0.75           | 0.688         |
|TXpower="-1dBm",beaconFraction=0.2| 0.25           | 0.063          | 0.75          |
|TXpower="-1dBm",beaconFraction=0.5| 0.938          | 0.313          | 0.938         |
|TXpower="-1dBm",beaconFraction=0.8| 1              | 0.063          | 1             |
+----------------------------------+----------------+----------------+----------------+
```

If we also use the –n switch then the resulting table will have a label (and dimension) named "node" and might also have a label (and dimension) named "index". These label names can also be used in with the --order switch. If not –n is used, and no order is specified, the node label is used as the columns of the table by default.

For valuePropagation, running our CastaliaResults with the –n switch and showing the "got value" output means that we can see which individual node got the output and which did not. Let's try it (we will not show the results here since the lines get too wide to be shown properly, you are encouraged to try the commands though and see the results for your self).

```
valuePropagation$ CastaliaResults -i 100812-102156.txt -s got -n -f -5dBm
```

Notice the use of the –f switch that filters the rows shown. In the example above only rows containing the string "-5dBm" will be shown (that is, 9 out of the 18 possible scenarios).

You notice that the lines of the output are long, since we have 16 nodes (i.e. 16 columns to show). You can change the order that the dimensions are displayed, but this might not help a lot. You can also try the more compacted output format using the switch –o 2.

```
valuePropagation$ CastaliaResults -i 100812-102156.txt -s got -n -f -5dBm -o 2
```

The compacted output format is primarily used so you can easily copy the results table to an Excel spreadsheet , and use the character 'l' as the cell separator. In a future version of Castalia we will also provide an output type that will automatically create figures with the help of Gnuplot.

What if we want to filter the rows even further. Say we want to have only rows that have beaconFraction=0.2 and TXPower=-5dBm. The –f switch accepts regular expressions (as does the –s wsitch). So we can give the following command:

```
valuePropagation$ CastaliaResults -i 100812-102156.txt -s got -n -f 0\.2.*-5dBm -o 2
```

So we gave the regular expression "0\.2.*-5dBm" which translates to: match any string that has 0.2 ('.' is a special character so we had to prefix it with '\') then has any number of characters (the '.*' part), and finally has the string '-5dBm'. You can find plenty material online on how to construct regex to achive a certain match.

3.4.2 Multiple repetitions and confidence intervals

You probably noticed that the value propagation results we got from our valuePropagation simulations did not seem to have any clear trends among the dimensions we used. The most clear trend was that with increasing beaconFraction we increase value propagation. However, if we were to draw the results, they would see too noisy or random. How about the other output of interest: Energy?

```
/valuePropagation$ CastaliaResults -i 100812-102156.txt -s energy
ResourceManager:Consumed Energy
+-----------------------------------+-----------------+-----------------+
|                                   | TXpower="-5dBm" | TXpower="-1dBm" |
+-----------------------------------+-----------------+-----------------+
| beaconFraction=0.2,dutyCycle=0.02 | 0.089           | 0.091           |
| beaconFraction=0.2,dutyCycle=0.05 | 0.106           | 0.106           |
| beaconFraction=0.2,dutyCycle=0.1  | 0.136           | 0.138           |
| beaconFraction=0.5,dutyCycle=0.02 | 0.096           | 0.11            |
| beaconFraction=0.5,dutyCycle=0.05 | 0.107           | 0.109           |
| beaconFraction=0.5,dutyCycle=0.1  | 0.137           | 0.141           |
| beaconFraction=0.8,dutyCycle=0.02 | 0.122           | 0.135           |
| beaconFraction=0.8,dutyCycle=0.05 | 0.118           | 0.106           |
| beaconFraction=0.8,dutyCycle=0.1  | 0.142           | 0.144           |
+-----------------------------------+-----------------+-----------------+
```

Trends seem more clear here. But how do we improve the results for the value propagation? How do we get to see clear trends? The randomness of results implies a randomness in the process that produced them. Indeed there are many random processes at play here: shadowing in the wireless channel, different start up times for the nodes, several MAC random decisions. Castalia has 11 distinct random number streams that affect different parts of the simulation. As a user you probably want to run your simulations with more that one set of random seeds (i.e., more that one set of produced random number streams). We can very easily do this by using the –r switch followed by the number of repetitions we want to run (each repetition is run with a different set of random seeds). Let's run 3 repetitions of the simulation we tried before:

```
valuePropagation$ Castalia -c varyDutyCycle,varyBeacon,varyTxPower -r 3
```

Have a look at the results:

```
valuePropagation$ CastaliaResults -i 100812-132444.txt -s energy
ResourceManager:Consumed Energy
+-----------------------------------+-----------------+-----------------+
|                                   | TXpower="-5dBm" | TXpower="-1dBm" |
+-----------------------------------+-----------------+-----------------+
| beaconFraction=0.2,dutyCycle=0.02 | 0.089           | 0.089           |
| beaconFraction=0.2,dutyCycle=0.05 | 0.107           | 0.107           |
| beaconFraction=0.2,dutyCycle=0.1  | 0.137           | 0.136           |
| beaconFraction=0.5,dutyCycle=0.02 | 0.092           | 0.12            |
| beaconFraction=0.5,dutyCycle=0.05 | 0.113           | 0.115           |
| beaconFraction=0.5,dutyCycle=0.1  | 0.139           | 0.141           |
| beaconFraction=0.8,dutyCycle=0.02 | 0.103           | 0.118           |
| beaconFraction=0.8,dutyCycle=0.05 | 0.118           | 0.126           |
| beaconFraction=0.8,dutyCycle=0.1  | 0.14            | 0.144           |
+-----------------------------------+-----------------+-----------------+
```

Energy results have not changed much compared to running just one simulation, which probably means that they are not affected by the random processes in Castalia. If one has a deeper understanding of the TunableMAC works and how energy is consumed, he would understand this observation to be indeed true. How about the value propagation results though?:

```
/valuePropagation$ CastaliaResults -i 100812-132444.txt -s got

Application:got value
+-----------------------------------+-----------------+-----------------+
|                                   | TXpower="-5dBm" | TXpower="-1dBm" |
+-----------------------------------+-----------------+-----------------+
```

```
| beaconFraction=0.2,dutyCycle=0.02 | 0.167           | 0.167           |
| beaconFraction=0.2,dutyCycle=0.05 | 0.104           | 0.292           |
| beaconFraction=0.2,dutyCycle=0.1  | 0.104           | 0.208           |
| beaconFraction=0.5,dutyCycle=0.02 | 0.146           | 0.979           |
| beaconFraction=0.5,dutyCycle=0.05 | 0.563           | 0.688           |
| beaconFraction=0.5,dutyCycle=0.1  | 0.563           | 0.958           |
| beaconFraction=0.8,dutyCycle=0.02 | 0.271           | 0.667           |
| beaconFraction=0.8,dutyCycle=0.05 | 0.667           | 1               |
| beaconFraction=0.8,dutyCycle=0.1  | 0.458           | 0.958           |
+-----------------------------------+-----------------+-----------------+
```

The results are quite different compared to the ones when we executed only one simulation. Another thing to notice, is that the results are "smoother", there is less extreme variation. Are 3 repetitions enough though? We do not want to run too many repetitions since simulation will take longer to complete, but how do we know a certain number is enough? We can use statistical tools to answer this question. More specifically we can instruct CastaliaResults to calculate the confidence intervals of the results over the repetitions it executed. To do that, we use the –c switch ('c' standing for 'confidence'), giving it the confidence level (level given as a percentage) we want our intervals calculated for:

```
valuePropagation$ CastaliaResults -i 100812-132444.txt -s got -c 90
Application:got value
(table omitted for brevity)
+-----------------------------------+-----------------+-----------------+
Application:got value - confidence intervals
+-----------------------------------+-----------------+-----------------+
|                                   | TXpower="-5dBm" | TXpower="-1dBm" |
+-----------------------------------+-----------------+-----------------+
| beaconFraction=0.2,dutyCycle=0.02 | 0.013           | 0.013           |
| beaconFraction=0.2,dutyCycle=0.05 | 0.011           | 0.016           |
| beaconFraction=0.2,dutyCycle=0.1  | 0.011           | 0.014           |
| beaconFraction=0.5,dutyCycle=0.02 | 0.012           | 0.005           |
| beaconFraction=0.5,dutyCycle=0.05 | 0.017           | 0.016           |
| beaconFraction=0.5,dutyCycle=0.1  | 0.017           | 0.007           |
| beaconFraction=0.8,dutyCycle=0.02 | 0.015           | 0.016           |
| beaconFraction=0.8,dutyCycle=0.05 | 0.016           | 0               |
| beaconFraction=0.8,dutyCycle=0.1  | 0.017           | 0.007           |
+-----------------------------------+-----------------+-----------------+
```

A 90% confidence interval CI, means that the values of the result are within the span [average-CI .. average+CI] with probability 90%.

In our particular example we see that the confidence intervals are not very low. If we take the 95% confidence interval (which is the most commonly used in network simulations) we see that the confidence intervals are unacceptably high:

```
valuePropagation$ CastaliaResults -i 100812-102156.txt -s got -c 95
Application:got value - confidence intervals
+-----------------------------------+-----------------+-----------------+
|                                   | TXpower="-5dBm" | TXpower="-1dBm" |
+-----------------------------------+-----------------+-----------------+
| beaconFraction=0.2,dutyCycle=0.02 | 0.049           | 0.055           |
| beaconFraction=0.2,dutyCycle=0.05 | 0.042           | 0.031           |
| beaconFraction=0.2,dutyCycle=0.1  | 0.031           | 0.055           |
| beaconFraction=0.5,dutyCycle=0.02 | 0.061           | 0.031           |
| beaconFraction=0.5,dutyCycle=0.05 | 0.031           | 0.059           |
| beaconFraction=0.5,dutyCycle=0.1  | 0.049           | 0.031           |
| beaconFraction=0.8,dutyCycle=0.02 | 0.049           | 0               |
| beaconFraction=0.8,dutyCycle=0.05 | 0.055           | 0.031           |
| beaconFraction=0.8,dutyCycle=0.1  | 0.059           | 0               |
+-----------------------------------+-----------------+-----------------+
```

You can look at the energy confidence intervals and notice how much lower they are. Let's run 15 repetitions to reduce the confidence intervals for value propagation:

```
valuePropagation$ Castalia -c varyDutyCycle,varyBeacon,varyTxPower -r 15
Running configuration 1/1
valuePropagation$ CastaliaResults -i 100812-134938.txt -s got -c 95
Application:got value
+-----------------------------------+-----------------+-----------------+
|                                   | TXpower="-5dBm" | TXpower="-1dBm" |
+-----------------------------------+-----------------+-----------------+
| beaconFraction=0.2,dutyCycle=0.02 | 0.146           | 0.25            |
| beaconFraction=0.2,dutyCycle=0.05 | 0.138           | 0.204           |
| beaconFraction=0.2,dutyCycle=0.1  | 0.154           | 0.538           |
| beaconFraction=0.5,dutyCycle=0.02 | 0.242           | 0.963           |
| beaconFraction=0.5,dutyCycle=0.05 | 0.292           | 0.829           |
| beaconFraction=0.5,dutyCycle=0.1  | 0.4             | 0.708           |
| beaconFraction=0.8,dutyCycle=0.02 | 0.729           | 0.813           |
| beaconFraction=0.8,dutyCycle=0.05 | 0.796           | 0.921           |
| beaconFraction=0.8,dutyCycle=0.1  | 0.679           | 0.992           |
+-----------------------------------+-----------------+-----------------+
Application:got value - confidence intervals
+-----------------------------------+-----------------+-----------------+
|                                   | TXpower="-5dBm" | TXpower="-1dBm" |
+-----------------------------------+-----------------+-----------------+
```

```
| beaconFraction=0.2,dutyCycle=0.02 | 0.003           | 0.004           |
| beaconFraction=0.2,dutyCycle=0.05 | 0.003           | 0.003           |
| beaconFraction=0.2,dutyCycle=0.1  | 0.003           | 0.004           |
| beaconFraction=0.5,dutyCycle=0.02 | 0.004           | 0.002           |
| beaconFraction=0.5,dutyCycle=0.05 | 0.004           | 0.003           |
| beaconFraction=0.5,dutyCycle=0.1  | 0.004           | 0.004           |
| beaconFraction=0.8,dutyCycle=0.02 | 0.004           | 0.003           |
| beaconFraction=0.8,dutyCycle=0.05 | 0.003           | 0.002           |
| beaconFraction=0.8,dutyCycle=0.1  | 0.004           | 0.001           |
+-----------------------------------+-----------------+-----------------+
```

We see a dramatic reduction in the confidence intervals. These are definitely in acceptable levels. You will also notice that the average results of "got value" reported are much smoother, following much clearer trends. Checking the confidence interval of your results, and executing more repetitions if needed, is a crucial part of simulation.

3.4.3 Body Area Network simulation example

Go `to Simulations/BANtest/` to see a quite complex configuration file (having 20 Config sections) that it used to simulate a variety of scenarios to evaluate MACs in Body Area Networks. In the General section you can see that we are using a custom pathLoss map which is derived from our experimental measurements and defined in a file `pathLossMap.txt` We also use a file to define the temporal variation of the wireless channel also derived from our measurement campaigns (section 4.1 provides more details about the format and meaning of these files).

All scenarios use the `throughputTest` application, where all nodes send packets to a sink/hub node at a constant (configurable) rate. Have a look at the summary of the configurations:

```
Castalia/Simulations/BANtest$ Castalia
List of available input files and configurations:
* omnetpp.ini
        General
        ZigBeeMAC
        GTSon
        GTSoff
        noTemporal
        BaselineMAC
        pollingON
```

```
        pollingOFF
        naivePolling
        minScheduled
        maxScheduled
        varyScheduled
        varyRAPlength
        oneNodeVaryRate
        oneNodeVaryPower
        oneNodeVaryTxNum
        allNodesVaryRate
        setRate
        setPower
        allNodesVaryPower
        varyReTxNum
```

You can see that there are two basic MACs: we can try Zigbee and BaselineMAC. We can vary the packet rate, the Tx power, the retransmission attempts, in one node or all the nodes. We can vary parameters of the MACs relating to scheduled access, random access and improvised access (polling). We can also choose a wireless channel with no temporal effects. We could run a simulation testing how Zigbee performs in a temporal vs non temporal channel for varying traffic rates.

```
BANtest$ Castalia -c ZigBeeMAC,allNodesVaryRate[General,noTemporal]
Running configuration 1/2
Running configuration 2/2
BANtest$ CastaliaResults -i 100813-013333.txt
+------------------+----------------------------------------+------------+
|           Module |                                 Output | Dimensions |
+------------------+----------------------------------------+------------+
|       Application |        Application level latency, in ms | 1x1(31)    |
|                   |                         Packets received | 1x5        |
| Communication.MAC | Fraction of time without PAN connection | 5x1        |
|                   |              Number of beacons received | 5x1
|                   |                  Number of beacons sent | 1x1        |
|                   |                        Packet breakdown | 5x1(5)     |
|Communication.Radio |                        RX pkt breakdown | 6x1(5)     |
|                   |                               TXed pkts | 6x1        |
|   ResourceManager |                         Consumed Energy | 6x1        |
|   wirelessChannel |                Fade depth distribution | 1x1(14)    |
+------------------+----------------------------------------+------------+
```

We see all the rich output produced by the simulation. Let's take a small peek:

```
BANtest$ CastaliaResults -i 100813-013333.txt -s received

Application:Packets received
+------------+---------+---------+---------+---------+---------+
|            | rate=12 | rate=14 | rate=16 | rate=18 | rate=20 |
+------------+---------+---------+---------+---------+---------+
| General    | 588.8   | 686     | 774.4   | 863.8   | 888.6   |
| noTemporal | 579.8   | 680.4   | 778.8   | 858.2   | 930.4   |
+------------+---------+---------+---------+---------+---------+


Communication.MAC:Number of beacons received
+------------+---------+---------+---------+---------+---------+
|            | rate=12 | rate=14 | rate=16 | rate=18 | rate=20 |
+------------+---------+---------+---------+---------+---------+
| General    | 101.2   | 99.8    | 101.2   | 100.4   | 100     |
| noTemporal | 103.6   | 103.2   | 104     | 103.6   | 103     |
+------------+---------+---------+---------+---------+---------+
```

Have a look at the breakdown of packets at the MAC level (too wide to show here):

```
BANtest$ CastaliaResults -i 100813-013333.txt -s "Packet break"
```

You are welcome to explore the performance of the protocols under different conditions. Just note that the Baseline MAC can sustain much higher packet rates, so be sure to use some of the commented higher rates lines.

3.4.4 Bridge Test simulation example

This simulation scenario was created to test some basic aspects of a WSN used for structural health monitoring of a bridge. It has the following characteristics:

The sensing nodes are placed in a grid throughout the simulation field within 20 meters from each other. The sink node is located in the middle of the field, and all sensing nodes will try to deliver their reports to the sink. Every 1 minute (on average) a car appears in the simulation field (CarsPhysicalProcess is used, which creates objects moving in lines). A passing car is guaranteed to trigger all sensing nodes along its path, thus creating a traffic flow towards the sink node in the network. Additionally, the sink node will be distributing several packets [totalling 5Kbytes] to all sensing nodes at the beginning of the simulation and then repeat doing so every day of the simulation. These packets symbolize an update software patch.

The goal of this scenario is to evaluate the performance of such WSN with different parameters in MAC and Routing layers, as well as different sizes of the bridge being monitored. Open the omnetpp.ini file and look at the configurations. Notice how we are creating the various deployments for different bridge sizes:

```
[Config 100mBridge]
SN.field_x = 100
SN.field_y = 20
SN.deployment = "[0]->center;[1..18]->6x3"
SN.numNodes = 19

SN.physicalProcess[0].point1_x_coord = 0
SN.physicalProcess[0].point1_y_coord = 10
SN.physicalProcess[0].point2_x_coord = 100
SN.physicalProcess[0].point2_y_coord = 10
```

4 Modeling in Castalia

This chapter explains how Castalia models the different aspects of a wireless sensor network from communications to physical process. It also gives a detailed account of all the modules and their parameters. This chapter is your reference to understand how to use the different modules and what you can do with them. The single common thing that we can mention for all modules in Castalia is that they all have a parameter called `collectTraceInfo`. This is set by default to 'false'. If set to 'true', the module will produce trace information that will be written in the Castalia-Trace.txt file. We have already seen examples of this in Chapter 3.

Communications is the most carefully modeled aspect of the Castalia simulator. From the wireless channel to the radio behavior and the implementation of different MAC protocols, we tried to capture the essence and many details of their real counterparts.

4.1 The Wireless Channel

The wireless channel is a notoriously difficult medium to model, especially when taking into account mobile nodes, a changing environment (e.g., in the BAN case: the body moving) and broadband communications. Even though there is a big corpus of theoretical results and experimental measurements for wireless communication in general, they tend to practically affect areas with more commercial impact such as cellular communication, and recently WiMAX. The mobile ad hoc networks and wireless sensor networks have not gained as much from the modeling advances of wireless communication. There are multiple reasons for this, the main ones being: 1) more complicated problem, 2) comparatively smaller immediate tangible commercial benefits to allow for the expensive and tedious job of detailed measurements and modeling, and 3) researchers coming mostly from the CS, networking background, lacking deep knowledge and interest in the physical layer. Nevertheless, some researchers in the community have taken up the much needed role and have produced models that fit experimental data, at least for some important aspects. The recent interest with Body Area Networks and the upcoming IEEE BAN standard has created another important momentum. For example, at NICTA, we have created experimental testbeds to capture hundreds of thousands measurements. Those measurements are analyzed to provide accurate models for both the average path losses around the body, and more importantly, the temporal variation behaviour of the channel.

We are confident to claim that, concerning the wireless channel, Castalia is the most realistic simulator one could find for WSN and BAN. The word "realistic" here is used in the sense that the simulator is making the necessary provisions to capture various important

features of the wireless channel. In order to have results that accurately reflect reality one would need appropriate input parameters and input files. In simple words, one needs to "tune" the simulator right. In the simulation examples provided with the distribution of Castalia, many of the parameters are set at reasonable values that reflect some real situation, some though –especially large input files modeling temporal variation– are not freely available and are NICTA's intellectual property. NICTA is interested in various forms of collaboration, so you are welcome to contact us regarding more accurate input files.

4.1.1 Average path loss modeling

One important aspect of the wireless channel modeling is to estimate the average path loss between two nodes, or in general, two points in space. For WSN, where the separation of nodes is from a couple of meters to a hundred meters, the lognormal shadowing model has been shown[5] to give accurate estimates for average path loss. This is the formula that returns path loss in dB as a function of the distance between two nodes and a few parameters

$$PL(d) = PL(d_0) + 10 \cdot \eta \cdot \log\left(\frac{d}{d_0}\right) + X_\sigma$$

PL(d) is the path loss at distance d, ***PL(d_0)*** is the known path loss at a reference distance ***d_0***, ***η*** is the path loss exponent, and X_σ is a gaussian zero-mean random variable with standard deviation ***σ***. The four parameters in italics and bold, are defined as parameters of the wireless channel module. Looking into src`/wirelessChannel/WirelessChannel.ned`

```
double pathLossExponent = default (2.4);
double PLd0 = default (55);
double d0 = default (1.0);
double sigma = default (4.0);
```

To access these parameters you have to prefix their name with "`SN.wirelessChannel.`"

The lognormal shadowing model is not very accurate if you want to capture the correlation between the two directions of a link[6]. If you treat the two directions as independent links, the variance you get is much larger than the one experienced in reality. For this reason we are using the model to return an average path loss for *both* directions of a link and then we add and subtract *a separate* Gaussian zero-mean random variable with

[5] Zuniga et. al.[1] have shown that the observed Packet Reception Rates (PRR) in experimental setups with mica2 motes can be explained with the lognormal shadowing channel model **and** appropriate models for the radio modulation (which Castalia uses).

[6] A link here is just a pair of nodes or more generally two points in space.

standard deviation `SN.wirelessChannel.bidirectionalSigma`. This standard deviation should generally be small (default = 1.0). This parameter replaces the old parameter SN.wirelessChannel.allBidirectionalLinks, which was a much coarser attempt to capture correlation between the two directions of a link. Now, with the new parameter we have a more clean and independent way to control correlation between the two directions.

If we are concerned with BAN modeling, the lognormal shadowing model does not produce good results. In this case, we can use another option that Castalia gives us and that is to explicitly set our path loss map. For example, we might have measured average path losses using a testbed (as we do at NICTA) and provide these as input to Castalia. This is done through the `SN.wirelessChannel.pathLossMapFile` parameter which gives the filename of the specially formatted input file.

The input file has lines of the following format:

`TxNodeID>RxNodeID1:dB_value,RxNodeID2:dB_value,…`
example: `0>1:56,2:40,3:59,4:54,5:58`

This means that when node 0 is transmitting, node 1 is experiencing 56dB path loss , node 2 is experiencing 40dB loss, node 3 59dBm loss, etc.

Before Castalia 3.0 we also provided another option: to give a map of packet reception probabilities (also called packet reception rates: PRR) using the `SN.wirelessChannel.PRRMapFile` parameter. This is no longer an option because of changes in the radio model. Now the radio can dynamically change its modulation type, plus we can have multiple kids of radios operating in a network. To translate a PRR to a path loss in dB, you need to know the operation parameters of the radios involved. If you can have different and dynamically changing radio operating parameters it makes little sense to determine a wireless channel with a PRR map. Thus, the `PRRMapFile` parameter the is now obsolete.

4.1.2 Allowing for node mobility

Allowing for mobile nodes, complicates matters, since now it is not enough to take the average path losses between the nodes. We need to keep state about path losses between points in the space. This implies that we need to break up the space in discrete cells and calculate the path losses from each cell to each other cell. Look at figure 4 for an example of such a map only for one transmitting cell.

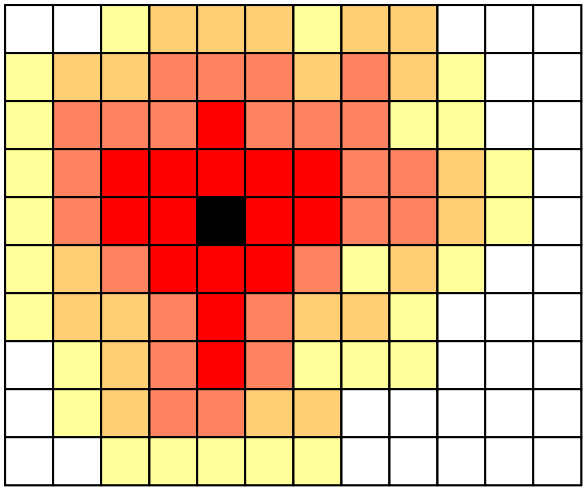

Figure 4: Path loss map in a 2D space segmented in cells (for a single transmitter cell)

The map is calculated with the model or input files we described before, we just take cell locations and cell IDs instead of specific node locations and node IDs. The parameters `SN.wirelessChannel.xCellSize`, `SN.wirelessChannel.yCellSize`, `SN.wirelessChannel.zCellSize` set the cell size (default = 5m). The smaller it is, the more fine-grained and accurate is your path loss map but also the more memory is needed to store it. Imagine you just have a 2-dimentional field 100m by 100m and you would like to have a cell of 1m y 1m, this results in 10,000 cells. This in turn results in $10{,}000^2$ = 100,000,000 path losses. With several bytes per path loss element (we store information besides the path loss) this can mean GBytes of memory. Our algorithm is smart and does not keep all possible combinations, but you get the idea on how fast can the state-space explode and also the processing time searching through these cells. Adding a 3rd dimension can aggravate matters, so be aware of the values you give to these parameters. In the wireless channel trace messages there is information about the size of the path loss map and the time it took to initialize it. This information can help you decide whether your choice of cell size with respect to the field is memory- and computation- efficient. Another parameter related to mobility in the wireless channel is `SN.wirelessChannel.onlyStaticNodes`. If we do not have any mobility in our scenario then we can declare this parameter true and save a lot of memory space and computation time. In this case, the space is not broken up into cells (the cell parameters do not matter, but still the need to be set to something) and the exact node locations are used in path loss computations much like Castalia 1.3 and below did.

Generally in Castalia, the modeling and computations are based on cells. When we do not have mobility, the nodes are treated as special cells where their location is acquired in a different manner, but all the rest of the code remains the same.

4.1.3 Temporal variation modeling

Another very important aspect of the wireless channel is the temporal variation. This is especially pronounced in rapidly changing environments as those experienced in a BAN. Figure 5 shows a typical profile of channel variation. Notice the sharp drops (measured up to -50dB in our BAN experiments) and that most of the time the channel is below the average path loss (0dB) (e.g., 2/3 of the time in Weibull fading channels). There are several theoretical models to describe temporal variation, taking their names from corresponding complex distributions (e.g., Rayleigh, Weibull, Nakagami, Gamma, lognormal). From our measurements around the human body we have seen that no single model describes temporal variation best. For that reason we tried to keep our modeling general but still expressively powerful to describe any temporal variation the measurements were showing.

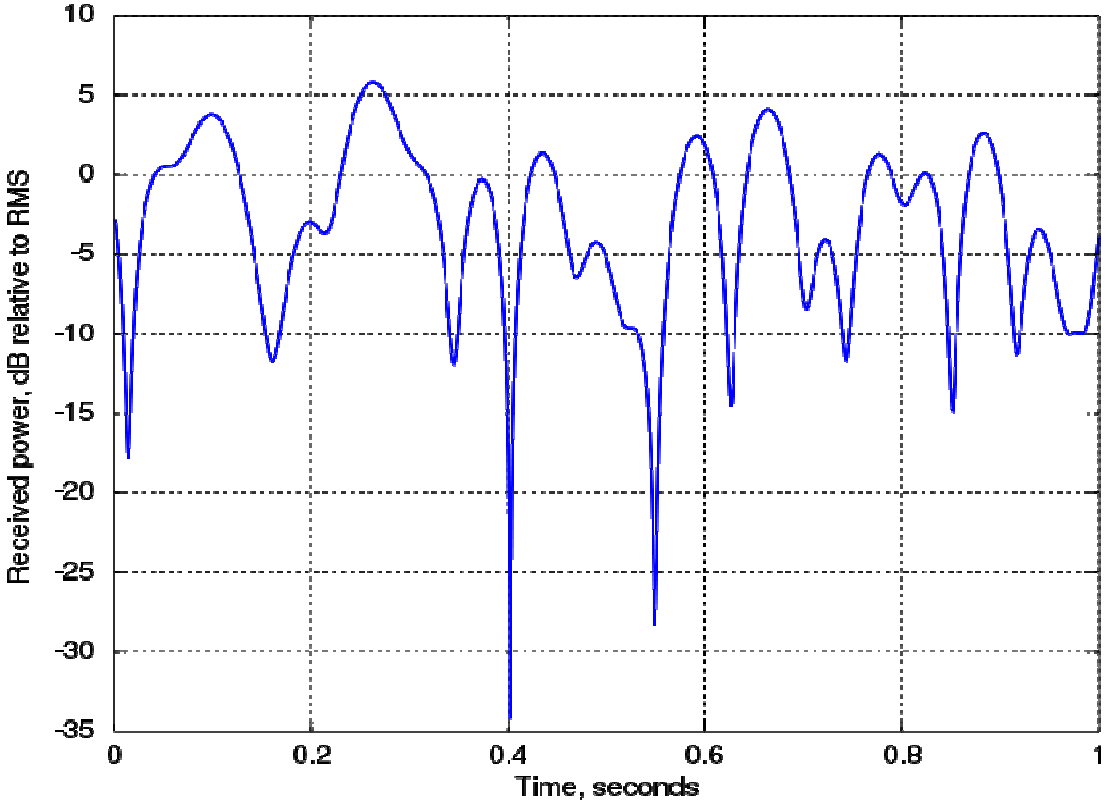

Figure 5: Typical temporal fading in wireless

In Castalia when we have to calculate the path loss at a specific moment, we first find the average path loss from the state we have stored during channel initialization (i.e., building the path loss map either using the model or the input files, as described in section 5.1.1) and then we have to find the component of the path loss due to temporal variation. To do this we

have kept the last "observed" value (in reality the last value we computed) and the time it has passed since then. These two numbers should define a probability density function (pdf) that we draw our new value from. Obviously, if little time has passed, then the bulk of the probability in this pdf should be in values close to the last observation. If the last observation is a deep fade (e.g. -40 dB) then the pdf should "boost" bigger values. The problem is that we cannot produce those pdfs dynamically from a model. They have to be produced from our experimental measurements. This in turn implies that we cannot have a pdf for any combination of last observation (dB value) and time passed. Instead we have to declare for what dB values and what times we are providing the pdfs. Then find a way to describe the pdfs in a manner that is accurate enough and computationally cheap. All this is done in a specially formatted file. The Castalia .ini file parameter is -as you have probably guessed- the name of that file: `SN.wirelessChannel.temporalModelParametersFile`

The file is formatted in the following way: (strings min_value, step, max_value, t1, t2, tn, and T should be replaced with floating point numbers. The expression{pdf description} is explained shortly after.

```
Signal variability (dB):  min_value:step:max_value
Correlation times (msec): t1,t2,…tn
Coherence time (msec): T
T: {pdf description}
tn,min_value: {pdf description}
tn,min_value+step: {pdf description}
tn,min_value+2step: {pdf description}
…
tn,max_value: {pdf description}
tn-1,min_value: {pdf description}
…
tn-1,max_value: {pdf description}
…
t1,max_value: {pdf description}
```

So the file describes the values and times it will provide pdf descriptions for, and then gives these pdf descriptions. It also gives an extra time which we call "coherence time" (in lieu of a better name) and an extra pdf for it. This is a large time, beyond which, it does not matter what was the last observed value and we just draw our number from its corresponding pdf.

Now, what is the pdf description? It is a generic way to describe any pdf (in a quantized way) that also makes drawing a number from it computationally fast. In its simplest form is an array of dB values, say 100 values, space-separated. To draw a number, we pick a random number between 1 and 100, and then we draw the value with the same index. In that way we can arbitrarily describe any pdf (within some accuracy). For example, let's take a smaller array with just 10 values (thus when drawing a value we pick a random number between 1 and 10): `23 -23 -23 -23 -23 10 10 10 -5 15`. This describes a pdf where value -23 has probability 0.5, value 10 has probability 0.3, and 0.1 probability for values -5 and 15. Now you understand that we are loosing some accuracy because of the limited number of values we have (if we have 100 values we are limited to probabilities of 0.01). The tails of the distributions we are concerned with are important though, so we need to capture their values which correspond to really small probabilities. We could increase the number of values we have to 1000 or 10,000 and allow for many duplicate values, but this would be a waste of memory. There is a more elegant way to do it that incurs an almost insignificant increase in computation time. We can define levels of draws. So at the top level we may have 10 buckets (usually filled with values), which all represent 0.1 probability. One of these buckets instead of having a value it can have a letter, that points is to the next level. If we the random number we pick chooses the letter, then we have to draw again from another array of values/letters. So if the second level has 5 values then each of these values has 1/10 * 1/5 = 0.02 probability. An example looks like this:

```
23 -23 -23 -23 -23 10 10 10 -5 A; A= 15 15 15 3.5 3.5
```

This defines a pdf very similar to the example before but now value 15 has probability 0.06 and there is a new value 3.5 with probability 0.04.

This is the general expression for {pdf description} ([] means optional):

```
pdf:= level; [letter= level;] ... [letter= level]
level:=  number ... number [letter] ... [letter]
```

There is a limit to the depth of levels that we can define, but for most practical reasons there is no point in going beyond level 3 or 4. In the temporal parameters file we give with the `BANtest` scenario we have taken a range -31:1:11, times of 1000ms, 250ms, 10ms which results in 129 pdfs. Our pdfs have 2 levels with 101 buckets in the first level and 10 in the second. We also have coherence time = 5secs and a 2 level pdf with 1001 buckets in the first level and 10 in the second. The total file size is 118Kbytes.

4.1.4 Delivering signals to the Radio module

In Castalia versions earlier than 3.0, the wireless channel was responsible for calculating the interference a receiver "sees", calculate the SINR (signal to interference plus noise ratio) and based on radio parameters decide whether a packet will be successfully received. If yes, then it would send the packet to the radio. So the wireless channel module was delivering packets to the radio of nodes. This modeling worked fine but it did not allow for multiple radio modes to co-exist in a network. Furthermore, it made fine-grain dynamic interference calculation challenging. Since Castalia 3.0 and the complete restructuring of the radio model, we were able to revisit the operation of the wireless channel too. Now the wireless channel just sends the signal to the radio and lets the radio calculate its interference, its (dynamically changing) SINR and decide whether or not it receives a packet. So now the wireless channel just needs to send messages to the radio that carry information about the signal such as: modulation, bandwidth, carrier frequency, and most importantly, the strength of the signal in dBm. Based on the path loss (average path loss + temporal variation) and the transmission power of a transmitter, the wireless channel can calculate the signal strength received at a node.

Which signals do we send to a radio module? All of them, irrespective of how weak they are? That would be a waste of resources; both computation time and memory. A signal transmission would result to a signal reception in all existing nodes irrespective of how far they are from the sender. Moreover if we have mobility, one cell would affect all other cells, resulting in the biggest possible path loss maps. Instead we can be smarter and set a signal strength threshold that beneath it does not make sense to deliver a signal. What would this threshold depend on? Obviously, the radio sensitivity. Should it just be the radio sensitivity? No, because a signal can be weaker than the radio sensitivity but it can still cause interference to another signal. Imagine two signals being heard at a receiver, one is at the sensitivity threshold, the other is 1dB smaller. If we did not deliver the smaller signal, the first one would be processed and the packet that it carried would be received with no problems. If we deliver both signals, a collision would probably occur. So how much lower than the radio sensitivity should we go? 10dB is usually more than enough and 5dB should be the minimum. Because there are tradeoffs at play we decided to make the signal delivery threshold a parameter of the wireless channel.

```
double signalDeliveryThreshold = default (-100);
```

-100dBm is a good value for both the BAN radio and the CC2420 we have defined. If you are using CC1000 then you should probably choose -103 or lower.

4.1.5 Choosing naive models

It would be useful to define simpler models (up to the rather simplistic disk model) so that simulation results can be compared with results from other simulators and more importantly so that the user can test different hypotheses on why a distributed algorithm does not work (or underperforms) when more realistic assumptions are taken into account . This is especially true with models of communication, especially the lower-level ones (channel and radio).

In an omnetpp.ini file, we can make the wireless channel simpler by simply setting:

```
SN.wirelessChannel.sigma = 0
SN.wirelessChannel.bidirectionalSigma = 0
```

This way all nodes at a certain distance from a transmitter get the exact same signal strength and all links are perfectly bidirectional (i.e., the quality of A→B is the same as the quality of B→A).

If we can also provide sharp thresholds for the radio reception (i.e., perfect reception of a packet or no reception at all) then we end up with the simple unit disk model. Indeed, for all the radios we have defined, we provide a special mode with an ideal modulation scheme. To choose it, simply set:

```
SN.node[*].Communication.Radio.mode = "IDEAL"
```

So by setting the two sigmas of the wireless channel to 0 and the radio mode to "IDEAL" a user can emulate the naïve -but still prevalent- unit disk communication mode (i.e., transmissions within a certain range from a transmitter are perfectly received, and outside this range not received at all).

When adopting a unit disk model, a user an control the range of the disk by controlling the SN.`wirelessChannel.PLd0` parameter. Assume you need the range of the disk to be 50m then PLd0 = (TxPowerUsed_dBm – max(receiverSensitivity, noisefloor+5dBm)) – 10*pathLossExponent*log(50) [7]

The user can further play with the `collisionModel` parameter of the Radio module to get simpler models, but we will explain this in the next section.

[7] In the radios we have defined so far SN.node[*].networkInterface.Radio.receiverSensitivity is greater or equal to SN.node[*].networkInterface.Radio.noiseFloor +5dBm

4.2 The Radio

The radio module tries to capture many features of a real low-power radio, one that is likely to be used in wireless sensor network platforms. The main features of our Radio module are:

- Multiple **states**: transmit, receive/listen, multiple (configurable) sleep states
- Transition **delays** from one state to another.
- Multiple (configurable) **transmission power levels**
- Different **power consumption** for the different states and Tx levels used.
- Multiple **modes of operation** (defined by modulation, datarate, bandwidth, noise floor, and other parameters) that can dynamically change.
- Multiple **modulation** schemes natively supported (FSK, PSK, DiffQPSK, DiffBPSK, Ideal modulation with 5dB threshold).
- Ability to define **custom modulation** schemes by defining SNR→ BER mapping
- Continuous calculation of **RSSI** (Received Signal Strength Indicator)
- **CCA** (Channel Clear Assessment) capability by interrupting the MAC module.
- Fine-grain **interference** calculation (dynamically changing during packet reception) and calculation of exact **bit errors** in a packet

4.2.1 The Radio Parameters file

To define the main operating parameters of a radio we are using a file which follows a specific format. We have already defined 3 radios that you can find in `Simulations/Parameters/Radio/`. These are: `BANRadio.txt` `CC1000.txt` `CC2420.txt`. BANRadio describes the radio proposed in the IEEE 802.15 Task Group 6 documents. CC2420 and CC1000 define the real radios of the same name by Texas Instruments. The parameter `RadioParametersFile` of the Radio module points to this file. The module will read it and parse it. Here is the format of a Castalia Radio Parameters file:

Any line beginning with '#' is considered a comment.

The files should contain 5 sections each starting with the line

RX MODES

TX LEVELS

DELAY TRANSITION MATRIX

POWER TRANSITION MATRIX

SLEEP LEVELS

The sections can appear in any order. In the radios we have defined we follow the order shown above. Section RX MODES contains one or more lines of the following format:

Name, dataRate(kbps), modulationType, bitsPerSymbol, bandwidth(MHz), noiseBandwidth(MHz), noiseFloor(dBm), sensitivity(dBm), powerConsumed(mW)

All quantities are numbers except from Name, which is an arbitrary string and modulationType which can take only the following values: FSK, PSK, DIFFBPSK, DIFFQPSK, IDEAL, CUSTOM. Here's an example from CC2420.txt:

```
normal, 250, PSK, 4, 20, 194, -100, -95, 62
```

Section TX_LEVELS should have two lines. The first line starting with the string "Tx_dBm" followed by n numbers (space-separated). The second line starting with the string "Tx_mW" also followed by n numbers separated by at least one space. As you can understand, the first line lists the output power of the different transmission levels in dBm and the second line how much energy the radio is spending when transmitting in a power level. The order of the power levels should be kept the same in both lines. In the radio files we have already defined, we chose to go from larger to smaller. Here's an example from CC2420.txt defining the 8 possible transmission power levels:

```
Tx_dBm 0 -1 -3 -5 -7 -10 -15 -25
Tx_mW 57.42 55.18 50.69 46.2 42.24 36.3 32.67 29.04
```

Section DELAY TRANSITION MATRIX specifies delays (in msec) to switch between the three main radio states: RX (receive), TX (transmit) and SLEEP. This section should contain a line corresponding to (and starting with the name of) each state. Following the name of a state there are three numbers defining transition times in milliseconds from RX, TX and SLEEP states respectively to the current state given first in each line. Below is an example from CC2420.txt:

```
RX     -     0.01  0.194
TX     0.01  -     0.194
SLEEP  0.05  0.05  -
```

In this case, it takes the radio 194μsec to change from SLEEP state to either RX or TX, 10μsec to change between TX and RX states and 50μsec to enter the SLEEP state. Note that each line contains '-' instead of a number in the cells corresponding to no transitions.

Similarly to the previous section, POWER TRANSITION MATRIX section follows the same format to specify the consumed power (in mW) during the time when a state transitions take place. In CC2420.txt configuration it can be seen that transitions to SLEEP state consume 1.4mW while any other transitions take 62mW:

```
RX      -     62    62
TX      62    -     62
SLEEP   1.4   1.4   -
```

Lastly, section SLEEP LEVELS defines different sleeping modes (or levels) available to the radio. Sleep levels exhibit different power consumption. The sleep levels are ordered from the most "shallow" sleep (i.e., the one with the more power consumption) to the deepest sleep. If in a sleep level, the radio can only go to the sleep level immediately up or down. You have to be in the top (shallowest) level to enter RX or TX state. When going from RX to TX you first go into the shallowest level and gradually go to a deeper sleep level if desired. This section should contain at least one line of the following format:

Name, power(mW), delay up(ms), power up(mW), delay down(ms), power down(ms)

Name is any string to denote the name of the level and power is the power consumption at that level. Delay up and delay down denote time taken to move to the level above and below (which respectively are defined by lines above and below), while values of power up and down correspond to the energy consumption value during the transition process. In CC2420.txt only a single sleep level is defined as follows (note that no transition values are given, since there are no other sleep levels):

```
idle 1.4, -, -, -, -
```

4.2.2 Radio Module Parameters

Apart from the Radio Parameters file, the radio module defines a set of parameters that can be specified from within configuration files (.ini). The list of these parameters can be found in the Radio.ned file and is presented below:

```
string RadioParametersFile = default ("");
```

This is a path to the radio parameters file as explained in the previous section. The path can be absolute, or relative to the directory which contains the simulation configuration (.ini) file.

```
string mode = default ("");
```

This parameter allows us to select the starting RX mode from the list defined in the radio parameters file. Default value (empty string) means that the first mode listed will be used. Modes can by changed dynamically during the simulation as will be explained in section 4.2.4.

```
string state = default ("RX")
```

This parameter specifies the starting state of the radio once simulation begins. By default the radio will start in the listening (receiving) state.

```
string TxOutputPower = default ("");
```

This parameter defines the starting transmission output power level to be used, e.g. -5dBm. Note that only power levels that were declared in the radio parameter file can be used. Default value (empty string) will cause the highest (first) output power level to be used.

```
string sleepLevel = default ("");
```

This parameter allows selecting the default sleep level (sleep depth) that will be used by the radio when a transition to SLEEP state is requested. The default value (empty string) means that the first level defined will be used. This corresponds to the fastest and most energy consuming sleep state.

```
double carrierFreq = default (2400.0);
```

This parameter allows selecting the starting carrier sense frequency of the radio (in MHz).

```
int collisionModel = default (2);
```

This parameter defines the collision model used by the radio to compute the impact of various incoming signals on each other's reception (this process is called interference). More information on this parameter and various available collision models will be provided in section 4.2.3

```
double CCAthreshold = default (-95.0);
```

This parameter defines the value of Clear Channel Assessment (CCA) threshold used by the radio when determining whether the wireless channel medium is clear. In particular, if the RSSI reading (i.e., an estimation of the cumulative power of incoming radio signals) exceeds this threshold, then the radio will report channel as busy (i.e. not clear).

```
int symbolsForRSSI = default (8);
```

This parameter specifies the amount of symbols required by the radio to perform RSSI calculation.

```
bool carrierSenseInterruptEnabled = default (false);
```

This parameter allows us to turn on an interrupt capability of the radio module which will inform upper layers (in particular MAC layer) when the channel state moves from clear to busy (i.e. when CCAthreshold is passed). This capability is present in some real radios, however it is turned off by default to ensure that carrier sense interrupt only appears when it is intended to be used.

4.2.3 Reception and Interference calculation

The Radio module operates on signals that are provided from the wireless channel module. Section 4.1.4 explained how signals are delivered. Signals last for some time since they are encoding a packet's transmission (OMNeT messages called `WC_SIGNAL_START` and `WC_SIGNAL_END` signify the duration of a signal/packet). Thus, at any time, the radio module has a list of incoming signals that are affecting each other's reception. If a signal is received without any other signals being received at the same time then the matter of its reception is decided thus: Based on the noise floor (a parameter of the RX mode of radio operation) and the received signal strength, we calculate the SNR. Based on the SNR, the modulation scheme (again, a parameter of the RX mode of radio operation) the data rate, and length of the packet/signal we calculate how many bit errors has the packet/signal experienced. Based on the encoding and bit errors we know if the packet was received correctly. When we add into the mix interfering signals then we need to calculate the Signal to Interference Ration (SINR or SIR) and do the above calculations at *every signal change*. Bit errors are kept up to the point of the last signal change. If a signal changes then we calculate SINR and bit errors for the last unchanged portion of the packet/signal.

How is interference calculated based on the reception of other signals? We simply add up the signal strengths of the interfering signals[8]. We also add up the power of the thermal noise (i.e. the noise floor) and we have the SINR.

The user has the option to opt for simpler interference/collision models by setting the parameter `SN.node[*].communication.radio.collisionModel`. If set to 0 *there are no collisions happening*. If set to 1 there is a simplistic model for collisions. In this case, if two nodes are concurrently transmitting and the receiver can receiver both of their signals -even minimally- then there is *always* a collision at the receiver. Notice that for the same case, with additive interference model, we can have two possibilities: a) have a collision as well or b) the receiver receives the stronger of the two transmissions (if it is strong enough). It will all depend on the specific SIR at the receiver node. Setting the collision

[8] Recent research has shown that the total interference is somewhat less that the simple addition of the interfering signals. In practice, you should not see any difference in the simulation unless you have a high-interference scenario, where one node is interfered with by 4 or more other nodes each time.

model variable to 2 uses the additive interference model where transmissions from other nodes are calculated as interference by linearly adding their effect at the receiver. In chapter 3 you have already seen the effect of using different collision models in the radioTest simulation scenarios.

4.2.4 Dynamically adjusting radio parameters

Section 4.2.2 illustrated a list of radio module parameters that allow to specify starting values of radio state, reception mode, sleep level, transmission power output, CCA threshold and carrier frequency. Values of these parameters can be set in the omnetpp.ini, but what if we would like to dynamically change the values for some of these parameters within a specific simulation run? After all, in real platforms, many or the radio parameters are controllable by the processor. For this purpose we have defined and implemented a series of functions to be used in the code of other communication modules (MAC and Routing) and application module, that allow changing the parameters of the radio.

First it is important to point out that communication between Castalia modules is handled using OMNeT messages. Consecutively, issuing a command to the radio module involves two steps: 1) create the correct control message (command) and 2) send it to the right module. Step one is largely automated using a set of pre-defined functions that return a command ready to be sent. These functions are defined in the 'RadioSupportFunctions.h' and are presented below:

```
RadioControlCommand
*createRadioCommand(RadioControlCommand_type,double)

RadioControlCommand
*createRadioCommand(RadioControlCommand_type,const char *)

RadioControlCommand
*createRadioCommand(RadioControlCommand_type,BasicState_type)

RadioControlCommand *createRadioCommand(RadioControlCommand_type)
```

It can be seen that all 4 functions return an object of RadioControlCommand type that can be transferred between OMNeT modules. Additionally, each function requires a RadioControlCommand_type argument. This type is defined in RadioControlMessage.msg and can take the following values:

```
SET_STATE

SET_MODE

SET_TX_OUTPUT

SET_SLEEP_LEVEL

SET_CARRIER_FREQ

SET_CCA_THRESHOLD
```

```
SET_CS_INTERRUPT_ON
SET_CS_INTERRUPT_OFF
```

Each of the four functions above is only compatible with a subset of the available command types, depending on the argument that needs to be provided. More specifically, double argument is required for SET_TX_OUTPUT, SET_CARRIER_FREQ and SET_CCA_THRESHOLD commands. String (char *) argument is needed for SET_MODE and SET_SLEEP_LEVEL commands, while BasicState_type value (corresponding to one of the 3 radio states – RX, TX and SLEEP) can only be used with SET_STATE command. Finally, no argument is required for SET_CS_INTERRUPT_ON and SET_CS_INTERRUPT_OFF radio control commands.

Moving on to the second step that involves forwarding command correctly. Realistically, there will be only two modules that will require interaction with the radio module: MAC and Application. In the Castalia architecture, MAC module is connected to the radio module since they are directly exchanging packets and messages. Consecutively, MAC module interface defined in VirtualMac.h file provides a helpful function `toRadioLayer()` that allows to send a message (or packet) to the radio. Below are some examples of controlling the radio from within MAC module's code:

```
toRadioLayer(createRadioCommand(SET_STATE,SLEEP));
toRadioLayer(createRadioCommand(SET_TX_OUTPUT,-5));
toRadioLayer(createRadioCommand(SET_CS_INTERRUPT_ON));
```

Application module on the other hand is only directly connected to Routing (Network) module, preventing it from sending messages to radio module directly. However, Castalia's communication stack has been designed in a way allowing to pass messages up and down the stack depending on their kind. In other words, Routing module will be able to accept a command designated to the radio module, but instead of trying to process it, the command will be forwarded down in the communication stack to the next layer below (MAC). Similarly, MAC module will forward the command further and deliver it to the radio module as required. Application module interface defined in VirtualApplication.h provides a `toNetworkLayer()` function that can be used to send radio control commands from within Aplication's code in the same way as from the MAC module's code:

```
toNetworkLayer(createRadioCommand(SET_STATE,SLEEP));
toNetworkLayer(createRadioCommand(SET_TX_OUTPUT,-5));
toNetworkLayer(createRadioCommand(SET_CS_INTERRUPT_ON));
```

It is important to note that it is uncommon to have both MAC and Application module controlling the radio at the same time. Moreover this can lead to incorrect operation of the whole communication stack. Therefore it is crucial to make sure that any commands issued by

the Application will not interfere with commands issued from the MAC module and vice versa.

4.3 MAC

The Medium Access Control protocol is an important part of the node's behaviour, thus there is a separate module that defines it. In Castalia we have fout main MAC modules implemented: 1) TunableMAC which exposes many parameters to the user and the application for tuning. From TunableMAC we also get the behaviour of a simple CSMA/CA MAC protocol (we have defined an ini file you can choose to include in your simulation). 2) the popular TMAC. From TMAC we can also get S-MAC by just setting a parameter (T-MAC is an enhancement of S-MAC allowing extendible active times). 3) IEEE 802.15.4 MAC. This is the standard for low power wireless networks, although *not* prevalent in WSN. 4) IEEE 802.15.6 MAC draft proposal for Body Area Networks (BAN)

4.3.1 Tunable MAC

Our initial motivation in building Castalia was to test a highly tunable MAC protocol in realistic channel radio conditions. We provide this highly tunable protocol with the standard distribution of Castalia because it can approximate several duty-cycling protocols out there (e.g., B-MAC, LPL). However it should be noted that this protocol was built with broadcast communication in mind (i.e., no unicast) thus it does not support acknowledgments, RTS and CTS control packets. The main function of the protocol is to duty cycle the radio and to transmit an appropriate train of beacons before each data transmission (since the nodes are not aligned in their schedules). It also provides several other parameterized functions such as retransmissions, probabilistic transmission, and backing off schemes. Look in the file

`src/node/communication/mac/tunableMac/TunableMAC.ned`

for a complete set of the module's parameters. Let us see the 9 most important parameters that can be tuned:

```
double dutyCycle = default (1.0);
```

This is the fraction of time that the node stays on listening to the channel. For (1-dutyCycle) fraction of the time the node sleeps. This is probably the most important parameter which affects energy consumption as it can drastically reduce listen time.

```
int listenInterval = default (10);
```

This is the time the node stays on listening. Knowing the duty cycle we can then define the time the node sleeps. After we had one listen and one sleeping interval, the cycle starts again. We would like the listen interval to be small so the sleeping interval for a given duty cycle is small too thus latency in data delivery is minimized. If we make the listen interval too small though packets won't be able to be received in full (either beacon or data packets). A good time for a listening interval is between 5ms and 10ms for the radio of 250Kbps we usually use. You should play around to find a good value for your scenarios.

```
double beaconIntervalFraction = default (1.0);
```

When we have a duty cycle we need a way to get the attention of a sleeping node before we transmit our data. A standard method is to use a train of beacons before our transmitted data. If we need to guarantee that a receiving node will wake up, we should make the train of beacons at least as long as the sleeping interval. But imagine you have many neighbors and you do not need all of them to wake up. Assume that you want (statistically) half of them to wake up. Then you should make the beacon interval half of the sleeping interval. This parameter expresses the *fraction* of the maximum beacon interval (= sleeping interval) that our beacon interval actually is. The smaller this is the less energy we spend, but the less chance we have to wake up a neighbor.

```
double probTx = default (1.0);
```

Probability of Transmissions. When a piece of data needs to be transmitted (or retransmitted) we perform the action with a certain probability; often assumed by many protocols to be 1. This value combined with the number of retransmissions can create any *expected number of transmissions per node*, even non integer values. For example if we have 6 retransmissions (so 7 transmissions in total) and probability of transmission 0.5, then we end up with 3.5 expected transmissions per node per data value we want to transmit.

```
int numTx = default (1);
```

Number of transmissions. For each piece of data needed to be transmitted how many times do we try to transmit it. Obviously the larger this parameter is the more energy is spent but more performance (reached nodes) can be achieved.

```
int randomTxOffset = default (10);
```

Random Transmission Offset. When we decide to transmit something we do not do it immediately but we wait a random time uniformly distributed in [0..Random Transmission Offset]. This randomness significantly helps to avoid collisions that could be very common in a broadcast type scenario. If a node transmits its data and four of its neighbors receive it and decide to retransmitted immediately then we are bound to have collisions. Adding the randomness in transmission time we can avoid such collisions. This value does not even have to be large provided that the nodes are performing carrier sense before transmitting (i.e., they are applying CSMA).

```
int reTxInterval = default (0);
```

Retransmission Interval. The interval between retransmissions. With this parameter we can effectively spread out our expected transmissions.

```
int backoffType = default (1);
```

This parameter has to do with carrier sensing. As we mentioned earlier, carrier sensing is performed normally by the MAC (the `carrierSense` parameter has to be "true"). This means that whenever the MAC needs to transmit a packet, and before starting transmitting potential beacons, it checks with the radio to see if the channel is clear. If the channel is not clear is the transmission is backed off for some time and the MAC puts the radio to sleep mode (to save energy). The backoff type parameter specifies how is this backing off interval determined. If set to 0, then we set the variable *backoffMaxInterval* to the duration of a sleeping interval. If set to 1 *backoffMaxInterval* is a constant time *backoffBaseValue*. If set to 2 then *backoffMaxInterval* depends on the consecutive *times* we found the channel not to be clear. This is the relation $backoffMaxInterval = (backoffBaseValue) \cdot (times)$. If we set the parameter to 3 then again it depends on the consecutive *times* we found the channel not to be clear: $backoffMaxInterval = (backoffBaseValue)^{times}$.

```
int backoffBaseValue = default (16);
```

This is the parameter we used above in expressing how long we are backing off for each different backing off type.

The rest of the parameters are defining how many bytes a data frame is, how much is the overhead, how large a beacon is, and an ack frame is (although not currently used). We also define the size of the MAC buffer and finally if the MAC is perrfomring carrier sensing.

4.3.1.1 Dynamically adjusting Tunable MAC parameters from the application module

As with the radio case, we would like to change many of the MAC parameters through the application code. We can send special messages/commands to TunableMAC to control it. This time we have not defined helper functions to create the commands so we have to be somewhat more verbose. The commands are defined in:

`src/node/communication/mac/tunableMac/TunableMacControl.msg`

Which means we have to add this line in the module code that wants to control TunableMAC:

```
#include “TunableMacControl_m.h”
```

Then to create a control command we do:

```
TunableMacControlCommand *cmd = new TunableMacControlCommand
                ("TunableMAC control command", MAC_CONTROL_COMMAND);
cmd->setTunableMacCommandKind (kind);
cmd->setParameter(value);
```

`kind` is an enum that can take the following values:

SET_DUTY_CYCLE

SET_LISTEN_INTERVAL

SET_BEACON_INTERVAL_FRACTION

SET_PROB_TX

SET_NUM_TX

SET_RANDOM_TX_OFFSET

SET_RETX_INTERVAL

SET_BACKOFF_TYPE

SET_BACKOFF_BASE_VALUE

`value` is a variable of double type.

After we create the command we send it in similar way we send radio commands. From an application module we call:

```
toNetworkLayer(cmd);
```

4.3.2 T-MAC and S-MAC

T-MAC is a popular MAC for WSN as it employs many techniques to keep the energy consumption low (using aggressive duty cycling and synchronization) while trying to keep performance (e.g. packet delivery) high by adapting its duty cycle according to the traffic needs. S-MAC can be seen as the predecessor of T-MAC as it initiated many of the techniques but uses a more rigid duty cycle. One Castalia module (TMAC) offers the functionality of both protocols. The implementation of T-MAC in Castalia was a complicated and time-consuming task since many of the protocol's practical details were not clearly described or explicitly stated in the initial paper. For one, all details about time-synchronisation were brushed under the carpet by referring to other works. More importantly, when the details of referenced techniques were found, they did not always made immediate sense to be applied as is in TMAC, so decision had to be taken to keep the performance of TMAC high. On top of this we added extra functionality that is not explicitly described in the paper such as a fixed number of retransmissions for failed unicast packets. The details of our implementation of TMAC, focusing on these uncertain points, are described in [6]. The parameters that the TMAC module take can be found in:

`src/node/communication/mac/tMac/TMAC.ned`

We start again with some parameters to control collecting traces and debug output. We then define the sizes of the different control packets in TMAC (sync, ack, rts cts) as well as the maximum frame size for data and its overhead. Also the buffer size of the MAC is given. Then 12 parameters follow that define the behaviour of the protocol. Let's see them in more detail. If you want to stay as close to the original TMAC leave these parameters to their default values.

```
int maxTxRetries = default (2);
```

This number of transmission attempts of a single unicast packet that TMAC will preform. A transmission is considered successful only if acknowledgment packet is received from the destination node. Sending an RTS packet is also considered as a transmission attempt. Note that this parameter does not apply to broadcast packets.

```
double frameTime = default (610);
```

The length of each frame period for all nodes (in milliseconds). Nodes try to synchronise the start and end of each frame with a global schedule (with the possibility of more than one schedules). Note that this refers to the duration of the whole frame; the active and inactive portions of each frame are determined dynamically and individually for each node.

```
double contentionPeriod = default (10);
```

The duration of contention interval (i.e. interval where transmissions of randomized), in milliseconds, for any transmission attempt. The major effect of this parameter is to avoid transmission interference from neighbouring nodes.

```
double listenTimeout = default (15);
```

The duration of listen timeout in milliseconds (can also be called activation timeout). This parameter defines the amount of time which has to pass without any activity on the wireless channel in order for a node to go to sleep in the current frame.

```
double waitTimeout = default (5);
```

The duration of timeout for expecting a reply from another node (in milliseconds). This reply may be a CTS packet or an ACK packet. If no reply is received after this time interval, then transmission attempt is considered failed and transmission attempt counter is decremented.

```
double resyncTime = default (6);
```

The interval between broadcasting synchronization packets (in seconds). The value of this parameter is directly related to the clock drift of nodes in the simulation network. Our experiments showed that 40 seconds is an adequate value to use with current clock drift model of Castalia. Note that original values in SMAC paper (which TMAC inherits) is 180.

```
bool allowSinkSync = default (true);
```

If this value is set to 1 (true), TMAC will attempt to extract information from higher layers (i.e., application module parameter `isSink`) in order to find out whether the current node is marked as 'sink'.

This parameter allows 'sink' nodes to avoid contention interval when creating a synchronisation schedule for the network, thus allowing for faster synchronisation, and consequently, better throughput (especially if packets need to be sent early in the simulation)

```
bool useFrts = default (false);
```

This parameter refers to the use of future request to send (FRTS) as defined by the creators of TMAC algorithm. In our current TMAC implementation we do not provide support for this

parameter as we consider it to be addressing a specific issue, while not being part of the core TMAC algorithm

```
bool useRtsCts = default (true);
```

This parameter is an additional feature, which was not originally proposed in TMAC. It allows to turn off RTS and CTS packets, thus limiting any transmission to a simple DATA - ACK exchange between nodes. We find this parameter useful for situations where only small packets are being sent, thus making it unnecessary to actually reserve the channel for transmission (since reservation will take more time than transmission itself).

```
bool disableTAextension = default (false);
```

Another extra parameter that allows us to turn off the activation timeout extension. Since the flexible active period is the trademark enhancement of TMAC over SMAC, by disabling it and defining appropriate listen interval (10% of the whole frame) we are essentially implementing SMAC. Indeed we provide an ini file to include in your simulations that defines the appropriate parameters of the TMAC module to emulate the SMAC protocol.

The final 2 parameters are defined from version 2.3 onwards and give greater flexibility that T-MAC does not explicitly define.

```
bool conservativeTA = default (true);
```

Use conservative activation timeout, will ensure that MAC stays awake for at least 15 ms after any activity on the radio. For more info read [6].

```
int collisionResolution = default (0);
```

Choose a collision resolution mechanism from those, described in [6]. Possible values are:
0 - Retry contention immediately after losing the channel,
1 - Retry only when heard a CTS or RTS
2 - Retry only in the next frame

If you want to use SMAC then the following file defines TMAC parameters in such a way as to emulalte SMAC. Include this file in your .ini file:
`Simulations/Parameters/MAC/SMAC.ini`

4.3.3 IEEE 802.15.4 MAC

The IEEE standard for wireless low-power short-range communications (802.15.4) defines, apart from the physical layer, the functionality of a MAC. We have implemented core functionality of this MAC in Castalia and we offer it starting with Castalia version 2.2. Castalia 2.3 included the important GTS functionality (a form of TDMA). We concentrated our implementation efforts in functions of the MAC that would help with Body Area Networks and left others unimplemented due to limited resources.

More specifically we implemented:

- CSMA-CA functionality (slotted and unslotted)
- Beacon-enabled PANs with association (auto associate)
- Direct data transfer mode
- Guaranteed time slots (GTS).

Features that are NOT implemented:

- Non-beacon PANs
- Indirect data transfer mode
- Multihop PAN topologies

We suggest you read the standard to have a better understanding of what the protocol does [7]. The brief description of its parameters here might not be adequate for full understanding if you have not studied the protocol before. The 802.15.4 MAC module has the usual communication module parameters (defining packet sizes and overhead) plus 17 protocol-specific parameters. Some of the important parameters that define durations of time are based in a variable called *aBaseSuperframeDuration* = aBaseSlotDuration * aNumSuperframeSlots * symbolTime. SymbolTime is derived by the radio parameters dataRate and bitsPerSymbol. The other two variables are parameters of the aforementioned 17 parameters and take on the default values 60 and 16 respectively. For the BAN radios we have currently defined in Castalia *aBaseSuperframeDuration* is 0.187msec.

Here are the 17 standard-specific parameters:

`enableSlottedCSMA`

Enables the slotted version of the CSMA algorithm, explained in the standard. We are not sure why a slotted version would be more desirable in assessing whether the channel is clear, and in our simulations it seems that an unslotted approach is marginally more desirable.

isFFD

Is the node a full function device? These are the only devices which can act as coordinators.

isPANCoordinator

Is the node the PAN coordinator?

batteryLifeExtention

If set to 1, it tries to reduce the randomisation offset in the slotted version of CSMA-CA algorithm.

frameOrder

Specifies how long is the active portion of a frame, when radios are listening or transmitting. Specifically it is equal to *aBaseSuperframeDuration* $\cdot 2^{\text{frameOrder}}$.

beaconOrder

Specifies how long is the full frame (active and inactive). This is the period between two beacons. Specifically it is equal to *aBaseSuperframeDuration* $\cdot 2^{\text{beaconOrder}}$.

unitBackoffPeriod

Specifies how long is the unit of time used in backing off. More specifically the standard uses a exponential backoff technique and the backoff time is: random$(1\ldots 2^{\text{backoff_exponent}}-1)$ · (unitBackoffPeriod · symbolTime). Backoff_exponent is a variable automatically audated by the protocol.

baseSlotDuration

The aforementioned parameter to calculate *aBaseSuperframeDuration*. Default is 60.

numSuperframeSlots

The aforementioned parameter to calculate *aBaseSuperframeDuration*. Default is 16.

macMinBE

In calculating the backoff time, this is the minimum value that the backoff exponent can take.

macMaxBE

In calculating the backoff time, this is the maximum value that the backoff exponent can take.

macMaxCSMABackoffs

Maximum number of backoffs until the transmission of the packet is aborted and go for another retry (if there are any left).

macMaxFrameRetries

Maximum number of retries until the packet is considered lost and the upper layer notified.

maxLostBeacons

Maximum number of beacons lost until the node considers itself disassociated from the PAN coordinator.

Castalia 2.3 introduced these 3 new parameters (relating to GTS)

minCAPlength =440

The minimum length of CAP period when GTS is used, defined in symbols

enableCAP =1

Allows a node to transmit DATA packets in the CAP period of a frame. Control packets are transmitted only in CAP regardless of this parameter's value

requestGTS =0

Allows a node to request a specified number of GTS slots from the coordinator. If the request is successful, DATA packets will be transmitted in the GTS slots assigned by the coordinator These slots are assigned dynamically according to availability. If no slots are available, the request will fail.

Note that this is an easy way for a node to statically request a certain number of slots. A more dynamic solution would be: the application module to instructs the MAC module how many slots should it request. The current (parameter-based) solution is a shortcut when we have constant application traffic and we can figure out a priory an optimal way to share the slots among the nodes. Indeed these are the kinds of simulations we are currently running with BAN scenarios.

Again you can refer to the standard [7] to have a better understanding of the parameters and the protocol. There is an .ini file in the `Parameter_Include_Files` directory that defines these parameters for the module.

4.3.4 Baseline BAN MAC

Body Area Networks is a fast evolving field of low-power wireless sensor networks. We put considerable effort in making Castalia a simulator suitable for BAN, and the MAC is one important area we focus. The BaselineBANMac module is our implementation of the IEEE 802.15.6 draft proposal for a standard in BAN MAC [8]. It is suggested you read the proposal to get an understanding of the MAC.

You can find the module description in:
`src/node/communication/mac/baselineBanMac/BaselineBANMac.ned`

The first 4 parameters defined are the usual MAC ones, defining buffer size, max packet size and mac packet overhead. Then we have 12 BANMAC-specific related parameters and 5 more parameters that are dependent on the PHY layer but the MAC needs to know about them, and has to define them as its own parameters. Here they are in detail:

```
bool isHub = default(false);
```

Is the node a hub (coordinator device)?

```
double allocationSlotLength = default(10);
```

Defines the length of the basic allocation slot in msecs

```
int beaconPeriodLength = default(32);
```

Defines the beacon period length in allocation slots

```
int RAP1Length= default(8);
```

Defines the Random Access Period length in slots

```
int scheduledAccessLength = default(0);
```

The slots asked by a sensor node from the hub to be assigned for scheduled access

```
int scheduledAccessPeriod = default(1);
```

If asking allocation slots for scheduled access, how often (in beacon periods) is the scheduled access requested for.

```
int maxPacketTries = default(2);
```

The maximum number of tries a packet will be attempted for successful reception.

```
double contentionSlotLength = default(0.36);
```

The length in msecs of the mini-slots used in contention

```
bool enhanceGuardTime = default(false);
```

The draft proposal suggests guard times to account for worse cases of de-synchronization among the nodes. However we discovered that while ending a TX we should really guard for 2GT instead of GT. This variable allows the more conservative guard time to be used.

```
bool enhanceMoreData = default(false);
```

If this variable is true then the moreData flag of the protocol carries information on how many more packets are waiting in the buffer to be transmitted, not just if there are more packets to transmit.

```
bool pollingEnabled = default(false);
```

A variable to control whether polling will be attempted.

```
bool naivePollingScheme = default(false);
```

A variable controlling how polling will be performed

The next 5 parameters are related to the PHY layer. As the proposal suggests we define them independantly.

```
double pTIFS = default(0.03);
```

The time in msecs, to start TXing a frame after you received one

```
double pTimeSleepToTX = default(0.2);
```

The time in msecs, to start TXing after being sleeping. NOT included in spec.

```
int phyLayerOverhead = default(6);
```

The overthead the radio adds in bytes

```
double phyDataRate = default(1024);
```

Radio's datarate in Kbps

```
double mClockAccuracy = default(0.0001);
```

The clock drift of the node's clock

4.4 Routing

Since version 1.2 of Castalia we have included a Network (routing) module. Initially routing was seen as a less important feature so no module was introduced to support it, if a user needed routing this had to be done at the application module. This was a result of our own needs: the algorithms we design do not use routing, since the nodes -which includes the node connected to the user/backhaul network- keep some state that gets updated by local communication messages only. The algorithms are designed in such a way so that the states converge. Nevertheless we acknowledge the general need for a modular routing structure thus we have introduced a network module since version 1.2. Version 1.3 brings the first implementations of simple routing protocols and it also updates some of the fields. There are two routing modules defined currently: `simpleTreeRouting` and the `multipathRingsRouting`. Both modules have several parameters that are common, such as the usual `moduleName` and `printDebugInfo` parameters, the `maxNetFrameSize` parameter, and parameters to set the length of control packets and overhead. `netBufferSize` specifies the size of the buffer found in the module (similar to the MAC and radio modules). `netSetupTimeout` specifies a timeout when the setting up of the network tree (or level information) is taking place. If a particular node does not receive timely information it might declare it self “not connected” and reports this to the application with a message.

Let us see how these two routing algorithms work. In the simpleTreeRouting a sink node has to initiate a tree_setup packet that gets propagated through the network. Nodes who receive this packet, name its sender as their parent. When a packet comes with sink as the destination they know to route it through the parent. The simpleTreeRouting module take some extra parameters to limit the possibilities of parent nodes. You can set a maximum number of neighbours and you can also set a RSSI threshold for a node to become your neighbour. Note that a parent can only be a neighboring node. The simpleTreeRouting is not supported in Castalia 3.0 (you have to use 2.3 if you need it). We found that this module needs major redesign so we deferred its re-implementation for version 3.1.

multipathRingsRouting is a little different in the sense that a node does not have a specific parent. A node just gets a level number (or ring number) during setup. The first setup packet sent from the sink has level 0. Any node that receives it adds 1 to the level and retransmit it. The process continues with every node adding 1 to the level of the received packet. Eventually all connected nodes will have a level number (there is still the possibility of unconnected nodes as before). When a node wants to send a packet to the sink it does not send it to a particular node but rather broadcasts it, attacking its level number. Any node with a smaller level number will rebroadcast it. The process continues until the sink is reached.

You can see with this algorithm many paths to the sink are taken. The algorithm is generally more robust unless there are many multiple and overlapping paths, in which case congestion becomes a problem.

How is a sink node defined? We could define a parameter in the routing module that gave the IDs of sink nodes. We have to question though: is this the job of the routing module? No. Whether or not a node is a sink node depends on the *application*. The application should know where the info should end up. Several applications already in Castalia are defining a parameter called `isSink`. The parameter takes values true/false and is defined for every node (as every node has an application module). The routing modules we have designed look at this application-level parameter to determine if the node is a sink node and whether the routing module in this particular node should start setting up the routing tree (or the routing rings). Thus, if you are using any of the two standard routing modules that Castalia comes with, your application should also define the `isSink` parameter. If you are defining your own routing module, you are of course free to change this dependency, but have in mind that routing modules normally do take input from the application.

With the two routing modules we also specified a generic format for the network frame. This includes header information: source (string), destination (string), and applicationID (string). A string was chosen to allow the most flexibility in defining destinations and sources. For example, routing protocols could have the following destinations: sink1, parent, point(23,56), areacircle(12, 35, 5), arearectangle(0,0 10, 40). The current routing protocols support only the destination “sink”.

4.5 The Physical Process

Sensing is usually grossly neglected in WSN simulators. The usual practice is to feed random numbers to nodes, or each node to have a static value. Issues like sensing device noise or bias are rarely taken into account. In Castalia we tried to go a couple of steps further than the usual practice and capture some essential elements of sensing.

The usual practice to sensed-data generation is to feed random numbers to nodes, or each node to have a static value, or at the best case feed the nodes with traces of sensed data. The last case is indeed realistic if we are concerned with a very specific physical process but the data traces rarely lend themselves to every kind of physical process simulation. For early phase algorithm design we need physical process models that are flexible enough yet have correspondence to real processes (e.g., spatial correlation of data, variability over time). For this purpose we created a generic physical process model in Castalia to feed the sensing devices of the nodes with data.

The basis of the model is sources of values that their influence is diffused over space. The sources can change in time and space, i.e., change their position and their value. The effect of multiple sources in a certain point is additive. More specifically the model that determines the value of the physical process at a certain location and at a certain time is:

$$V(p,t) = \sum_{all\ sources\ \ i} \frac{V_i(t)}{(K \cdot d_i(t) + 1)^a}$$

Where: $V(p,t)$ denotes the value of the physical process at point p, at time t

$V_i(t)$ denotes the value of the i^{th} source at time t

$d_i(t)$ denotes the distance of point p from the i^{th} source at time t

K, a are parameters that determine how is the value from a source diffused.

Parameters K and a are determined directly in the omnetpp.ini file by setting `SN.physicalProcess[0].multiplicative_k` and `SN.physicalProcess[0].attenuation_exp_a`

$V_i(t)$ and $d_i(t)$ are not directly given but they are calculated by the way we describe the behaviour of the sources. This is described by the parameter

`SN.physicalProcess[0].source_i` where i is the i^{th} source. This is a string in the form: "time pos_x pos_y value; time pos_x pos_y value; time pos_x pos_y value; …"

Each one of the tuples (time, pos_x, pos_y, value) is called a snapshot of the source. To help with array allocation issues the user should define the maximum possible number of snapshots for all sources using the `SN.physicalProcess[0].max_num_snapshots` parameter. The user also needs to define the number of sources in the physical process using the `SN.physicalProcess[0].numSources` parameter.

To get an idea of the complex and interesting behaviour you can model with this generic model you can view the following video http://castalia.npc.nicta.com.au/support/two_sources.avi . It shows the evolution of a physical process with two sources that are moving in the field and are changing their values out of phase.

Whereas this model is useful, there are times the user needs to have a much easier (and usually simple and static) control on the values fed to the sensing devices. For example, in the value propagation application we want one or a few nodes to get a value beyond a threshold. Although this can be done with our generic model, it requires some calculations in order to determine the right parameters. There is no need to burden the user in such a way. It would be much easier if the user could directly specify the values that nodes should take.

Acknowledging this need we allow the user to determine the values that the nodes are sensing (i.e., values taken from the physical process module) with the directly and statically. To do so we must set the `SN.physicalProcess[0].inputType` parameter to 0. If we set it to 1 then the generic physical process model will be chosen and the relevant parameters mentioned before will be interpreted. Setting it to 0 though, the output of the physical process will be determined solely by the `SN.physicalProcess[0].directNodeValueAssignment` parameter. This parameter is a string and its syntax is the following: "(default_value) nodeID_A:value_A nodeID_B:val_B …". That is, nodes that their ID is mentioned in the string are taking the value associate with this id, and nodes not mentioned are just taking the default value. For example, if we have the string "(0) 6:40" all nodes will receive the value 0 except node 6 which will receive value 40. Note here that these values are static, so the value returned is not dependent on the time the node asks a value from the physical process.

4.6 The Sensor Manager

The “ground truth” offered by the physical process is distorted by the inaccuracies of the sensing devices. In Castalia we offer a set of parameters to model this distortion. Before elaborating in the specifics of each parameter we would like to say a few words on how sensing devices are tied to the physical processes.

In Castalia, there is a one-to one correspondence between sensing device type and a physical process module. This might seem limiting as in reality a single physical process could trigger sensing devices of several modalities. For example, an explosion might trigger microphones, light sensors and temperature sensors. Furthermore, a sensor can be affected by multiple distinct physical processes (e.g., a temperature sensor can be affected by climatic conditions + a nearby fire). Although we acknowledge these interconnections we decided to go for a simpler representation and implementation hoping that it will cover most of the practical needs. You probably noticed in the description of the physical process model that we support only one output per point per time (i.e., one type of value, one sensing modality). Similarly there is no way for a sensing device to connect to more than one physical process module. So a more accurate name for the physical process module would be “overall physical process influence to sensing modality X” module. For most practical purposes this will represent an influence from a single physical process (e.g., a nearby fire will be much more important than climatic conditions to a temperature sensor). This one-to-one correspondence does not limit the user to one modality or one physical process, it just makes the code less modular when there are complex interactions.

After this long parenthesis we are ready to look at the parameters Castalia offers for the sensing device (all prefixed with `SN.node[*].nodeSensorDevMgr.`):

`numSensingDevices` the number of different sensing device types we have at a node.

`sensorTypes` string with the names for each of the different sensing device types (space separated)

`pwrConsumptionPerDevice` string-array with the power consumption per sample (in mJoules) for each sensing device (space separated).

`corrPhyProcess` string-array that holds the index of the Physical process that each one of the sensing device type corresponds (space separated).

`maxSampleRates` string-array that holds the maximum samples per sec rate for each if the sensing device types (space separated).

`devicesBias` string-array with the sigmas of the bias for each one of the sensing device types. So if the bias sigma is 3 for the temperature sensors then each time we instantiate a new temperature sensor we draw from a normal distribution with sigma 3 in order to find the constant bias of this particular device.

`devicesNoise` string-array with the sigmas of the noise for each one of the sensing device types. So if the noise sigma is 1 for the temperature sensors, each time we get a sample from a temperature sensor (not just when we instantiate one device as above) we add noise drawn from a normal distribution with sigma 1.

`devicesSensitivity` the minmum value the device can return (new with ver 2.1)

`devicesResolution` the resolution of discrete readings the device can return (new with ver 2.1)

`devicesSaturation` the maximum value the device can return (new with ver 2.1)

`devicesDrift` not implemented yet

`devicesHysterisis` not implemented yet

4.7 The Mobility Manager

The mobility module specifies how the nodes move through space. It holds location state that other modules can access at any time using a function call, and it also notifies the wireless channel periodically of the position of a node. The notification of the wireless channel is done for efficiency reasons. Since the wireless channel needs the location of all nodes very often (every time we have the start of a packet transmission or carrier sensing), it would be detrimental to speed if it had to explicitly ask for the locations of all nodes (only to find out that in most cases nothing changed). It is far better for the mobility module to notify the channel if something changed (for example it could notify it, when the node has just changed cells). However calculation of cell changes can be difficult and do not necessarily need to be part of the Mobility module, thus we give the much simpler (but still efficient) option or periodic updates.

We have defined a virtual mobility module that all other mobility modules should be derived from. It has two parameters:

```
SN.node[*].MobilityManager.collectTraceInfo
SN.node[*].MobilityManager.updateInterval
```

We also have an extra parameter in the node to define the name of the mobility module it is using (much like we do for the application module):

```
SN.node[*].MobilityManagerName
```

Currently we have only implemented one mobility pattern module, the simple `LineMobilityManager`. The user just describes the destination point of a line segment (starting point is the starting location of the node) and thus defines a trajectory for the node to move back and forth. The user also defines the speed that the node is moving. These are the parameters that do that:

```
SN.node[*].MobilityManager.xCoorDestination
SN.node[*].MobilityManager.yCoorDestination
SN.node[*].MobilityManager.zCoorDestination
SN.node[*].MobilityManager.speed
```

If you want to create your own Mobility module(s) with more advanced mobility patterns then read Chapter 5.

4.8 The Resource Manager

The resource manager module keeps track of various node resources the most important of all being energy. It is a rather special module in Castalia. Since most other modules need to communicate with it (e.g., to consume energy) the message passing method is not very efficient so we opted to directly call some functions of the resource manager. Currently, the functionality of the resource manager is very simple. It linearly subtracts the energy requested by different modules and keeps track of memory statistics. We have plans to include more accurate battery models, keep track of time used in computation and based on computation time and CPU power mode find the CPU energy consumption. Currently, most of the parameters related to the resource manager found in omnetpp.ini are not functional.

5 Creating your own Application, Routing, MAC, and Mobility Manager modules

So far you have seen and hopefully familiarize yourself with the basics of Castalia. You know how to run simulations, view the output, understand the functionality of different modules and their parameters. You can indeed create a wide variety of simulations based on the available modules. But what if you need to introduce your own distributed algorithm in Castalia by introducing your own application module? What if you want to implement a new MAC protocol or a new Routing protocol? How about implementing a Mobility Manager that support new mobility patterns? Castalia has provisions that make the creation of any of these 4 kinds of modules easier. We will start by diving general instructions applying to all 4 kinds of modules and then we will see some issues that related to each kind separately.

After you are done with all the new files creation, remember to rebuild Castalia using the following commands from the top-most Castalia directory.

```
Castalia $ make clean
Castalia $./makemake
Castalia $ make
```

If you are using any external libraries, consult the last section of the Installation Guide to find out how to include them in the build process.

The first step is to determine the correct location for the new module code within the Castalia directory structure and create a dedicated directory to hold the source code of the new module. The correct locations are:

for Application: `src/node/application/`

for Routing: `src/node/communication/routing/`

for MAC: `src/node/communication/mac/`

for mobility: `src/node/mobilityManager/`

Each of these directories already includes several subdirectories that contain various implementations of relevant modules. After a directory for the new module is created, it is necessary to define that module using the NED syntax (i.e. create the module's .ned file). This file is created inside the module's dedicated directory and is named using the new module's name. Assume that the new module to be created is called “NewCastaliaModule”. Following OMNeT’s naming convention; the dedicated directory for this module should be named

`newCastaliaModule` - i.e. starting with lower case letter. Consecutively, it is needed to create a file `NewCastaliaModule.ned`. In this file it is necessary to define the package of the module. The package can be obtained by taking the path to the .ned file relative to the Castalia's `src/` directory, and then replacing '/' symbols with '.'. For example, if NewCastaliaModule is a MAC module, then the package in NewCastaliaModule.ned file will be declared as:

```
package node.communication.mac.newCastaliaModule;
```

Next step is to define the module itself. Here we note that the module's parent directory contains a .ned file that has a name starting with the letter 'i'. This is the *interface file* that has to be followed in order for the new module to fit into the structure of Castalia. For NewCastaliaModule to be recognized as a Castalia application, it has to be declared using the 'like' keyword, referencing the interface module. For example:

```
simple NewCastaliaModule like node.communication.mac.iMac
```

Next comes a list of parameters that will be passed to the module at runtime from the simulation configuration. Each interface .ned file will already include a set of parameters that are mandatory for all MAC modules, however new parameters can be included. Below is a full example of NewCastaliaModule.ned file:

```
package node.communication.mac.newCastaliaModule;
simple NewCastaliaModule like node.communication.mac.iMac {
	parameters:
		bool collectTraceInfo;
		int macMaxPacketSize;
		int macBufferSize;
		int macPacketOverhead;

		int newParameter1;
		string newParameter2 = default("default value");
		bool newParameter3 = default(false);
	gates:
		output toNetworkModule;
		output toRadioModule;
		input fromNetworkModule;
		input fromRadioModule;
		input fromCommModuleResourceMgr;
}
```

The first four parameters are defined in the iMac.ned interface file and are mandatory. Each interface file will have its own set of mandatory parameters, that will vary depending on which module is being created. After the mandatory parameters for a MAC there are 3 parameters specific to NewCastaliaModule. Note that it is possible to provide default values for parameters (even for mandatory parameters from the interface). Gates are fully defined by the interface .ned file and have to match exactly (i.e. 'gates:' section can be entirely copied from interface file). This concludes the definition of the .ned file.

The next step is to write the actual code of the module. A module it represented by a C++ object, and it has to inherit from appropriate virtual classes that are provided. These virtual classes are the main help Castalia offers in defining new modules. A developer can use the methods and structure already defined by the virtual classes.

For our current example it is necessary to create the files NewCastaliaModule.h and NewCastaliaModule.cc. In the .h file we declare a new C++ class that will implement our module. The name of the class has to match the name of the module and the name of the .ned file. As we already mentioned, the class inherits from one of the virtual classes we provide according to the kind of module we are creating.

For Application: `class NewCastaliaModule : public VirtualApplication{`

For Routing: `class NewCastaliaModule : public VirtualRouting {`

For MAC: `class NewCastaliaModule : public VirtualMac {`

For MobilityManager: `class NewCastaliaModule : public VirtualMobilityManager{`

In the .cc file we first include the .h we just created and then we call the Define_Module macro to register our new creation as an OMNeT module:

```
Define_Module(NewCastaliaModule);
```

After this we have to define (or redefine) the methods that the virtual class declares/defines. The methods depend on the kind of module we are implementing and thus we will see them in separate sections

5.1 Defining a new Application module

The VirtualApplication class defines a base for any Castalia application. It provides the following virtual methods that can be optionally overwritten. You can look at the already implemented ThroughputTest app module to see how these methods are used.

```
void startup()
```

This method is called at initialization

```
void finishSpecific()
```

This method is called in the end of simulation

```
void handleSensorReading(SensorReadingMessage *)
void handleNetworkControlMessage(cMessage *)
void handleMacControlMessage(cMessage *)
void handleRadioControlMessage(RadioControlMessage *)
```

These methods allow us to react to control messages from various communication layers.

A virtual method that is mandatory for the new app module to define:

```
void fromNetworkLayer(ApplicationGenericDataPacket *packet, const
char *srcAddr, double RSSI, double LQI)
```

This function will be called when a packet is received from the communication stack.

Functions that are provided (i.e. can be called from the module's code):

```
void requestSensorReading(int index)
```

Request a reading from sensor with given index. The result will be returned with `handleSensorReading` function call

```
void toNetworkLayer(cMessage *msg)
```

Sends a control message to the communication stack

```
void toNetworkLayer(cPacket *pkt, const char *dstAddr)
```

Sends a data packet to the communication stack, to be delivered to given address

```
ApplicationGenericDataPacket *createGenericDataPacket(double data,
int sequenceNum, int size)
```

Creates a simple data packet of given size and with given sequence number

5.2 Defining a new Routing module

VirtualRouting defines a base for any routing module class by providing the following virtual functions that can be defined to change modules behavior:

```
void startup()
```

This function is called at initialization

```
void finishSpecific()
```

This function is called in the end of simulation

```
void handleNetworkControlCommand(cMessage *)
```

Reacts to control commands from the application layer

Virtual functions that are mandatory for the new Routing module to work:

```
void fromApplicationLayer(cPacket *pkt, const char *dstAddr)
```

Process a data packet received from the app (outgoing data packet)

```
void fromMacLayer(cPacket *pkt, int, double, double)
```

Process a data packet coming from the MAC (incoming data packet)

Functions that are provided (i.e. can be called from the module's code):

```
int bufferPacket(cPacket *pkt)
```

Stores an incoming data packet in the buffer

```
void toApplicationLayer(cMessage *msg)
```

Sends messages and packets to application layer

```
void toMacLayer(cMessage *msg)
```

Sends message to MAC layer

```
void toMacLayer(cPacket *pkt, int macAddr)
```

Sends a packet to MAC layer, MAC address has to be provided

```
void encapsulatePacket(cPacket *, cPacket *)
```

Encapsulates application packet in a routing packet

```
cPacket *decapsulatePacket(cPacket *)
```

Decapsulates routing packet to extract application packet

```
int resolveNetworkAddress(const char *)
```

Converts Network layer address to MAC layer address

It would be instructional to see implemented Routing modules on their use of these methods.

5.3 Defining a new MAC module

VirtualMac defines a base for any MAC class and provides the following virtual functions that can be overwritten to change a module's behavior:

```
void startup()
```

This function is called at initialization

```
void finishSpecific()
```

This function is called in the end of simulation

```
int handleControlCommand(cMessage * msg)
```

Allows to react to control commands from higher layers

Virtual functions that are mandatory for a new MAC module to work:

```
void fromNetworkLayer(cPacket *pkt, int dstAddr)
```

Processes packet received from the Routing layer (outgoing packet)

```
void fromRadioLayer(cPacket *pkt, double RSSI, double LQI)
```

Processes packet received from the Radio (incoming packet)

Functions that are provided (i.e. can be called from the module's code):

```
int bufferPacket(cPacket *pkt)
```

Stores an incoming data packet in the buffer

```
void toNetworkLayer(cMessage *msg)
```

Sends packet/message to Routing layer

```
void toRadioLayer(cMessage *msg)
```

Sends packet/message to Radio

```
void encapsulatePacket(cPacket *, cPacket *)
```

Encapsulates routing packet in a MAC packet

```
cPacket *decapsulatePacket(cPacket *)
```

Decapsulates MAC packet to extract routing packet

You can look at TunableMAC to see how these methods are used more concretely.

5.4 Defining a new Mobility manager module

VirtualMobilityManager class defines a base for any mobility manager module and provides the following virtual functions that can be defined to change a module's behavior:

```
void initialize()
```

This function is called at initialization

Functions that are provided (i.e. can be called from the module's code):

```
void notifyWirelessChannel()
```

Notifies the wireless channel about this node's location. Within the module itself it is possible to create a periodic self message that will update the node location every given time interval (see `LineMobilityModule` for example) or more complex patterns can be created if needed.

```
void setLocation(double x, double y, double z)
void setLocation(NodeLocation_type)
```

These methods alter a node's location and automatically notify the wireless channel.

The current node location can be accessed through the protected variable `nodeLocation` of type `NodeLocation_type`

6 References

[1] Draft Text Narrowband Physical Layer, https://mentor.ieee.org/802.15/dcn/10/15-10-0195-02-0006-draft-text-narrowband-physical-layer.doc

[2] http://focus.ti.com/docs/prod/folders/print/cc1000.html

[3] http://focus.ti.com/docs/prod/folders/print/cc2420.html

[4] Marco Zuniga, Bhaskar Krishnamachari, "Analyzing the Transitional Region in Low Power Wireless Links", First IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON), Santa Clara, CA, October 2004.

[5] Karim Seada, Marco Zuniga, Ahmed Helmy, Bhaskar Krishnamachari, "Energy Efficient Forwarding Strategies for Geographic Routing in Wireless Sensor Networks," ACM Sensys 2004, November 2004.

[6] Yuri Tselishchev, Athanassios Boulis, Lavy Libman, “Experiences and Lessons from Implementing a Wireless Sensor Network MAC Protocol in the Castalia Simulator,” submitted to IEEE Wireless Communications & Networking Conference 2010 (WCNC 2010), Sydney, Australia.

[7] The IEEE 802.15.4 standard (ver. 2006) http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf

[8] MAC and Security Baseline Proposal, https://mentor.ieee.org/802.15/dcn/10/15-10-0196-02-0006-mac-and-security-baseline-proposal-c-normative-text-doc.doc