Wireless Ad-hoc and Sensor Networks
COSC 418, Semester 2, 2015
Project Description

Dr. Andreas Willig

July 18, 2016

1 Summary

In this project you will explore some basic issues in wireless communications and networking, related to the physical layer and the medium access control layer. You will do so by developing a simulation model using the OMNet++ discrete-event simulation framework,[1] in which you have to add your own code using the C++ programming language [2]. Most likely you will have to familiarize yourself with OMNet++, and in some cases also with (a subset of) the C++ programming language.

You can use OMNet++ under Windows or under Linux, note that in both cases you have to build it yourself. Unfortunately, I am not in a position to support you on Windows. Furthermore, OMNet++ can be integrated with the Eclipse IDE, which I have no experience with. All the instructions given in the following (e.g. hints for the installation of OMNet++) assume that you are using Linux.

You will work in groups of two students. The marks for the project will be based on the achieved functionality of your code (as evidenced in a final report and individual conversations of 15-minutes duration) and your simulation results, the final presentation (during the lecture slot of the last week of term four) and the final report (to be submitted on the same day as the final presentation).

Important: I have tried my best to make the following description as clear, helpful and correct/consistent as I possibly can, but I suspect that there is room for improvement. If you have any question or any doubt please let me know (by email, on the learn forum or during the lecture). I will update this project description as necessary.

[1] `www.omnetpp.org`

2 Project Timeline

- During the third term you will acquaint yourself with the C++ programming language and the OMNet++ discrete-event simulation framework.
- In the third week of term three the project description (this document) will be published, and will be updated as necessary. You will be notified through learn about important updates.
- In the fourth term we will use some part of each lecture to discuss the projects.
- In the sixth week of the fourth term each pair will present its project results to other students. This presentation should take 10 - 15 minutes. I will also schedule with each individual a separate demonstration meeting (approximately 10 - 15 minutes) in which you will give me a brief walk through the source code. This meeting will take place in the sixth week of the fourth term. They will be scheduled at a later stage.
- Furthermore, each group needs to submit a **report** of five to ten pages (discussing their results, tests, and other things, see below) by the time of the final presentation.

2.1 C++

If you do not know already, you will notice that C++ is a complex language with many features, building on the C language. In the following I assume that you are already familiar with C (as you should after attending ENCE 260 or an equivalent course) and know the basics of include files, expressions, data types, control structures (`if`, `for`, ...), functions and pointers. The C++ language is described comprehensively in the book of Stroustrup [2] (but there are a lot of other books as well), and tutorial / reference material can be found in the web in many places. Use it.

In order to work with OMNet++ and for the purposes of this project you do not need to know the full C++ language. The important bits include:

- Namespaces
- Structures and classes
- Derived classes, inheritance and class hierarchies (but not necessarily multiple inheritance)
- Constructors, Destructors, Copy and Move
- Basics of operator overloading
- Memory management

2.2 OMNet++

All explanations refer to the current stable version of OMNet++, which is version 5.0. I strongly recommend you use this version.

2.2.1 Installation and Test

Please follow the following steps (I assume you are using the `bash` shell):

1. Download OMNet++ from the website `www.omnetpp.org`. Under Linux, you should download OMNet++ version 5.0 in the `.tgz` format (shorthand for `.tar.gz`, a `tar` file that has been `gzipped`). If you do the download from your browser, you will likely find the file in the `Downloads` sub-directory of your home directory. I will assume that the file is called `omnetpp-5.0-src.tgz`.

2. Move the file `omnetpp-5.0-src.tgz` to a place where you want to unpack and build it.

3. Unpack it, using the command:

   ```
   tar xvfz omnetpp-5.0-src.tgz
   ```

   After this you will have a new sub-directory called `omnetpp-5.0/` in your current directory. Change into it. From now on I assume you are there.

4. Look at the `README` file, and afterwards look at the installation instructions. You find these in file `doc/InstallGuide.pdf`. This guide contains installation instructions for all the major operating systems. Follow these and build OMNet++, perhaps after you have installed any additional packages required. Note that you might have to set the environment variable `TCL_LIBRARY`. On my system (a Ubuntu system) it is `/usr/lib/tcl8.6` for my particular `tcl` version. You should also follow the instructions for modifying your `.bashrc` file to add the `bin/` subdirectory of your OMNet++ installation to your `PATH` environment variable, and perhaps also set `TCL_LIBRARY` appropriately.

5. After building OMNet++, test your installation by giving the commands:

   ```
   cd samples/dyna
   ./dyna
   ```

   and you should see a graphical application coming up.

2.2.2 Reading Guidelines

The most important resource is the OMNet++ simulation manual, which you find on the website. I would also suggest to look at some of the tutorials, in particular the TicToc tutorial.

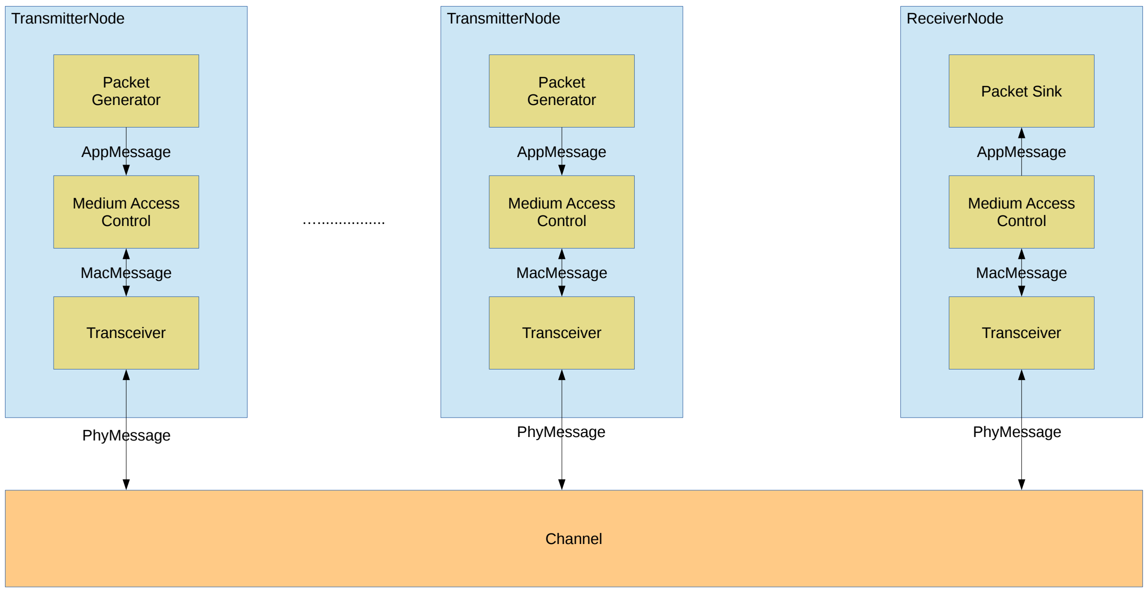

Figure 1: System Setup

You do not have to read the user manual cover to cover. You should read Chapters 1 to 6, 9 and 10 fully, all the other chapters you should skim to see what they contain and read more deeply when needed.

3 Project Description

In your project you will use OMNet++ to investigate the performance of a simple CSMA-type medium access control protocol over a wireless channel model. An overview over the considered system is shown in Figure 1. When in the following we speak of “higher layers” or “lower layers”, this is to be taken with reference to the figure – for example, from the perspective of the medium access control module the packet generator is “higher” and the transceiver is “lower”.

In our system we will have a variable number of **transmitter nodes**, which broadly want to send data packets to a single **receiver node** over a common and shared wireless channel. A transmitter node consists of the following components / modules:

1. The **packet generator** module generates application layer messages with some (stochastic) interarrival time distribution that can be chosen in the configuration file. These messages are sent to the MAC module of the same station.
2. The **MAC** (medium access control) module accepts messages to transmit from its higher layers, buffers these in a queue of bounded size (excess messages are to be

dropped) and executes a CSMA-type medium access control protocol to transmit these messages over the channel. To do this, it interacts with the local transceiver for transmission and reception of packets and for carrier-sense operations. In the other direction, any packets being received from the lower layers are eventually sent to the higher layers after minor processing.

3. The **transceiver** module actually transmits packets on the channel and helps the MAC with carrier-sensing. Furthermore, it receives packets from the channel and determines whether these packets are correctly received – if not, it will drop the received packet.

The receiver node will use the same MAC and transceiver modules, but instead of a packet generator module it will use a **packet sink** module, which accepts incoming packets (from its local MAC) and prints out some information about these (e.g. arrival time, identification of sending node, information about user data) in a logfile.

The **channel** module distributes packets / signals between all attached stations, most of the modeling of wireless propagation takes place in the transceivers.

All these modules are described in the following.

4 TransmitterNode

In OMNet++-terminology, a `TransmitterNode` is a compound module which has a `PacketGenerator`, a `MAC` and a `Transceiver` module as its child modules. Furthermore, the `TransmitterNode` module should have own parameters, including the following ones:

- A `nodeIdentifier`, which is an integer value used to identify the individual node. These identifiers should be set in the configuration file **and it is crucial that you ensure that different nodes get distinct identifiers**.
- The `nodeXPosition` and `nodeYPosition` parameters describe the geographical position of the node in two-dimensional space. An alternative would be to place these two parameters into the `Transceiver` module.

5 ReceiverNode

The `ReceiverNode` is also a compound module, which has a `Transceiver`, a `MAC` and a `PacketSink` module as its child modules. Furthermore, the `ReceiverNode` module has two parameters, the `nodeXPosition` and `nodeYPosition` parameters describing its geographical location (but again, these might go into the `Transceiver` when this is any easier).

6 Packet Generator

The `PacketGenerator` module (which would be a simple module in OMNet++ terminology) creates an infinite stream of new application layer messages (of message type `AppMessage`) and hands these over to the local `MAC` module. Any messages that the `MAC` might wish to hand over to the `PacketGenerator` shall simply be discarded. The inter-generation times are to be drawn from a random distribution that is given as a parameter (named `iatDistribution`) to the `PacketGenerator` module (see the OMNet++ manual for how to do that). A second module parameter is the `messageSize`, which shall indicate the size of the generated message in bytes (note that we do not actually generate messages of this size, we only use the size field later on to fix the packet length and derive the packet error probability from that). Furthermore, the `PacketGenerator` maintains a local integer variable called `seqno`, which is initialized to zero.

The `PacketGenerator` generates messages of type `AppMessage`. Such an `AppMessage` should contain at least the following fields:

1. `timeStamp`: simulation time at which the message has been generated.
2. `senderId`: is the node identification of the `TransmitterNode` sending this message – this has to be obtained from the parent module (which should be a `TransmitterNode` compound module).
3. `sequenceNumber`: the current value of the `seqno` variable is copied into this field and afterwards `seqno` is incremented.
4. `msgSize`: indicates the total message size in bytes, to be taken from the `messageSize` module parameter.

The underlying `MAC` will also send any received packets to this module. The `PacketGenerator` should simply drop them.

7 Packet Sink

The `PacketSink` module (a simple module) accepts messages of type `AppMessage` from its local `MAC` module, prints textual information about such a message to a log file, possibly records other statistics, and disposes the message. This module needs only one module parameter, which is the filename of the log file.

For each received message the `PacketSink` should write the following information to the log file:

1. The time at which `PacketSink` has received the message
2. The fields from the `AppMessage` type, i.e. the `timeStamp` (at which the message has been generated by the source node), the `senderId`, the `sequenceNumber` and the `msgSize`.

8 Transceiver

The `Transceiver` module is probably the most complex one, as most of the (very simple in our case) modeling of wireless channel properties takes place here. It takes packets from the `MAC` and models a transmission of this packet with a given transmit power and transceiver turnaround operations. On the receiving side, it tracks packet transmissions from all stations, checks for collisions, and for uncollided packets it calculates the signal-to-noise ratio (SNR) at the receiver, converts this into a bit error rate p and subsequently into a packet error rate (PER) P and drops the packet with probability P.

8.1 Module Parameters

The `Transceiver` module has the following module parameters:

- `txPowerDBm` specifies the radiated power in dBm[2]
- `bitRate` specifies the transmission bitrate in bits/s.
- `csThreshDBm` specifies the observed signal power (in dBm) above which a carrier-sensing station will indicate the medium as busy.
- `noisePowerDBm` specifies the noise power (which here we purport to be expressed in units of dBm, which is not entirely correct).
- `turnaroundTime` specifies the time required for turning the transceiver from the transmit state to the receive state or vice versa.
- `csTime` specifies the time required to carry out a carrier sense operation

8.2 Internal Variables

The `Transceiver` module has a number of internal variables, some of which I will describe here:

- The variable `transceiverState` can assume two different values: a receive state (in which the transceiver is ready to receive data or actually receiving data) and a transmit state, in which the transceiver is preparing for transmission or actually carrying it out. The default is to be in the receive state. The transmit state is only entered upon request from the `MAC` module and consists of three distinct stages:
  - After getting the request from the `MAC` the `Transceiver` changes to the transmit state, then waits for an amount of time given as the `turnaroundTime`, modeling the transceiver turnaround time.

[2]This unit specifies milliwatts in the decibel domain. A transmit power of 1 mW corresponds to 0 dBm.

  - Then actual packet transmission is handled between `Transceiver` and `Channel`. The time this takes is given by the packet length in bits divided by the `bitRate` parameter.
  - After packet transmission we wait for another `turnaroundTime` before switching back to the receive state.

  You will need to come up with a sensible decision on how the `Transceiver` behaves when it gets further requests from the `MAC` while being in transmit state.

- A variable `currentTransmissions` maintains a list of currently active transmissions (including own transmissions, if any) and for each transmission records important features like the identity of the sender and the received power.

This is probably not sufficient. I suggest you code the entire `Transceiver` as a finite state machine, for which you will require further variables.

8.3 Carrier-Sensing

As part of its operation, the `MAC` wants to perform carrier-sensing operations. In real systems, this is carried out by the transceiver and the carrier sense operation consumes some amount of time, as the channel signal is averaged over a certain time interval to reduce noise.

The `MAC` module starts the process by sending a message of type `CSRequest` (which does not need any own fields) to the `Transceiver`. When the `Transceiver` is in the receive state, it performs the following steps:

- it first calculates the current signal power (in dBm) observed on the channel (see below)
- then it waits for a time corresponding to the `csTime` parameter, and finally
- it sends a message of type `CSResponse` back to the local `MAC` module. The `CSResponse` message has one Boolean field called `busyChannel`. This field is set to `TRUE` when the current signal power observed in the first step exceeds the value of the parameter `csThreshDBm`, otherwise it is set to `FALSE`.

You need to decide how you react when your `Transceiver` receives a `CSRequest` message while being in the transmit state.

To calculate the current signal power (in dBm), you should traverse the list of current transmissions (`currentTransmissions`) and add up the received power of all ongoing transmissions in the normal domain, not in the dB domain, and finally convert the sum to dBm. This models a setup where the powers of all involved signals simply add up.

8.4 Transmission Path

For actual transmission, two different types of messages are exchanged between the `MAC` and the `Transceiver` modules:

- A message of type `TransmissionRequest` is sent from the `MAC` to the `Transceiver`. It encapsulates a MAC packet, which is contained in a message of type `MacMessage`.
- A message of type `TransmissionConfirm` is sent from the `Transceiver` to the `MAC` to indicate the outcome of the packet transmission. It has a single field called `status`.

When the `MAC` wants to transmit a packet, it sends a message of type `TransmissionRequest` to the `Transceiver`. When the `Transceiver` is already in the transmit state when the `TransmissionRequest` arrives, it should respond with a message of type `TransmissionConfirm` in which the field `status` is set to `statusBusy`, and the `Transceiver` takes no further action (except perhaps some logging).

When the `Transceiver` is in the receive state when getting the `TransmissionRequest`, it takes the following steps (see the discussion for the transceiver state above):

1. It retrieves the packet length n in bits from the MAC packet.
2. It changes its internal state `transceiverState` to the transmit state.
3. It waits for a time specified by the `TurnaroundTime` parameter.
4. After that it signals start of the transmission to the `Channel` by sending a message of type `SignalStart` (see below).
5. It awaits the end of the packet transmission (which requires time n/`Bitrate`), then it signals the end of the transmission to the channel by sending a message of type `SignalEnd`.
6. After that it waits for another `TurnaroundTime`.
7. It changes its internal state to the receive state.
8. It sends a message of type `TransmissionConfirm` to its `MAC` where the `status` field is set to `statusOK`.

The message of type `SignalStart` sent to the `Channel` module should contain at least the following information fields:

- `transmitPowerDBm` into which the `Transceiver` simply copies the value of its `txPowerDBm` parameter.
- `positionX` and `positionY`, into which the `Transceiver` simply copies the values of the `nodeXPosition` and `nodeYPosition` parameters of the surrounding module

of type `TransmitterNode` or `ReceiverNode`.[3]

- `identifier`, which contains the value of the `nodeIdentifier` parameter of the surrounding compound module.
- `collidedFlag`, which is a Boolean flag initialized to `FALSE`.
- The actual MAC message of type `MacMessage` is encapsulated as well.[4]

The message of type `SignalStop` sent to the `Channel` module contains only a field called `identifier`, into which the sending `Transceiver` copies the value of the `nodeIdentifier` parameter of its surrounding compound module.

The `Channel` module will copy the `SignalStart` and `SignalStop` messages to **all** `Transceiver` modules attached to the channel, including the `Transceiver` that sends these messages.

8.5 Receive Path

Broadly, the receive path of the `Transceiver` processes all the `SignalStart` and `SignalStop` messages coming from the `Channel`, independent of whether it is in the transmit or receive state. These messages are used to update the variable `currentTransmissions`, which is a list of all currently ongoing transmissions. In fact, the `SignalStart` messages themselves are stored in `currentTransmissions`. There should be at all times at most one ongoing transmission from any node, and you should check this condition in various places.

When the `Transceiver` gets a `SignalStart` message, it extracts its `identifier` field and checks whether it equals the `identifier` field of any of the `SignalStart` messages stored in the `currentTransmissions` list. If so, print an error message and abort the program. Otherwise, just add the `SignalStart` message to the `currentTransmissions` list. Furthermore, if at the time of arrival of the `SignalStart` message the `currentTransmissions` list was non-empty, then you should change the `collidedFlag` fields of the freshly received message and of all messages stored in `currentTransmissions` to `TRUE`, as a collision has happened on the channel.[5]

When the `Transceiver` gets a `SignalStop` message, it first extracts the `identifier` field and checks whether it equals one of the identifiers of the messages stored in the `currentTransmissions` list. If no such message is found, then an error message should be printed and the program aborted. Otherwise, denoting the corresponding message found in `currentTransmissions` as `msg`, we perform the following steps:

[3]As discussed above, it might also be sensible to place these parameters of the surrounding compound modules into the `Transceiver` instead.

[4]Look up the concept of message encapsulation in OMNet++.

[5]We will make the simplifying assumption that collisions always render the involved packets undecodeable, i.e. erroneous. This is not necessarily true in reality. Can you imagine situations where two packets collide but one of them can still be decoded?

1. Remove `msg` from `currentTransmissions` but retain it for the next steps.
2. If `msg.collidedFlag` is `TRUE`, then we simply drop `msg` and stop any further processing. By this, collided messages are always dropped.
3. Calculate the Euclidean distance δ between ourselves (with the x- and y-positions taken from the parameters `nodeXPosition` and `nodeYPosition` of the surrounding compound module) and the sender of the message `msg`, for which we extract its position from `msg`.
4. Calculate the path loss for distance δ using a path loss model with a reference distance d_0 of 1 m and assuming no path loss for distances smaller than d_0. For the path loss computation you should use a path loss exponent parameter `pathLossExponent`, which could be a global parameter of the simulation (or you put it into the `Channel` and have the latter copy this value into the `SignalStart` messages). Convert this path loss then to Decibel, giving $\widehat{\delta}$. More on these calculations below.
5. From the transmit power of the message (`msg.transmitPowerDBm`) and the calculated path loss $\widehat{\delta}$ calculate the received power (in dBm) and use this together with the `noisePowerDBm` parameter to calculate the bit error rate p (assuming simple BPSK modulation), and from the knowledge of p and the length of the received packet in bits, n, next calculate the packet error probability P. More on these calculations below.
6. Draw a uniformly distributed random number u between 0 and 1. If $u < P$, then stop any further processing and drop the packet as it is erroneous. Otherwise, extract the MAC packet out of `msg` (we refer to it as `mpkt` and it should be of message type `MacMessage`) and encapsulate it into a message of type `TransmissionIndication` that you send to the local `MAC` module.
7. Dispose of `msg`.

8.6 The BER calculations

In the following, all quantities in the dB domain are marked with a hat, like e.g. $\widehat{X}$, whereas quantities in the "normal" domain (X) do not have a hat. In doing this, we disregard the difference between dB (expressing a dimensionless ratio) and dBm (having dimension of mW).

As noted above, we first extract the transmit power in dBm from the message (we denote it by $\widehat{T}$). Next, we obtain the geographical position (x_0, y_0) of the transmitting node from the packet and our own geographic position (x_1, y_1) from our own parameters and calculate the transmission distance d as the Euclidean distance between these two points:

$$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

From this distance we calculate the path loss δ as follows:

$$\delta = \begin{cases} 1 & : \quad d \leq 1 \\ d^\gamma & : \quad d > 1 \end{cases}$$

where γ is the path loss exponent (taken from the corresponding parameter `pathLossExponent`). After conversion of δ into the dB domain (giving $\widehat{\delta}$) we can calculate the received power in dBm as

$$\widehat{R} = \widehat{T} - \widehat{\delta}$$

Next, if we denote by $\widehat{N}$ the value of the `noisePowerDBm` parameter and by $\widehat{r}$ the value of the `bitRate` parameter in the dB domain, we can calculate a quantity related to the signal-to-noise ratio[6] in the dB domain as

$$\widehat{E} = \widehat{R} - (\widehat{N} + \widehat{r})$$

We can finally calculate the bit error rate p as follows

$$p = Q\left(\sqrt{2 \cdot \left[\widehat{E}\right]^\vee}\right)$$

where $[x]^\vee$ is an operator that converts a number back from the dB domain to the normal domain,[7] and $Q(\cdot)$ is the well-known **complementary error function**, which is in turn defined as

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \exp\left(-\frac{u^2}{2}\right) du$$

Finally, it should be a straightforward matter to calculate the packet error rate P for a packet of length n bits and the given bit error rate p. You will need to figure out the formula.

A few comments about these calculations are in order:

- An attempt has been made to do the calculations in the dB domain until the very end. This is good practice as the numbers otherwise become rather large (or small) and numerical overflows / underflows might result.
- The complementary error function $Q(\cdot)$ is also known as the `erfc` function, and is available in C++ when you include the `<cmath>` and `<ctgmath>` header files.
- As a test, with $\gamma = 4$, $\widehat{T} = 0$, $\widehat{N} = -120$ and $r = 250,000$, I get the following results for a distance of $d = 50$ m:
  - $\widehat{\delta} = 67.9588$

[6] This quantity, in the normal domain, can be understood as the ratio of the energy spent per transmitted bit to the energy of the noise signal – communications people know this as the E_b/N_0 ratio, and this is equivalent to the signal-to-noise ratio.

[7] As you might remember, to convert a value v from the normal domain to the dB domain we calculate $\widehat{v} = 10 \log_{10} v$ and the inverse conversion is $v = [\widehat{v}]^\vee = 10^{\widehat{v}/10}$.

- $\widehat{R} = -67.9588$
- $\widehat{E} = -1.9382$
- $p = 0.1096$

where these numbers have been rounded / truncated.

- These calculations are incomplete and not entirely correct, in particular with respect to the noise modeling. A more correct version would have required further parameters and conversions.
- If you want to dig further: the bit error rate formula is valid for one particular and very simple modulation scheme called binary phase shift keying (BPSK). You find much more information in communications textbooks, like for example in [1].

9 Channel

The `Channel` module is much simpler than the `Transceiver`. It needs to be connected to all `Transceiver` modules present in the system, and has only one important task: it takes messages of type `SignalStart` or `SignalEnd` from any `Transceiver` and sends copies of these to *each* attached `Transceiver` (including the sending one). Make sure that you really create **deep copies** of these messages and clarify what you do with the message that you have received before. This is a place where you indeed need to watch for your memory usage.

10 MAC

The main function of the `MAC` module is to implement a non-persistent CSMA MAC scheme. There are no acknowledgements and no retransmissions. In this section I will deliberately be less precise, as I want you to figure out the details yourselves.

The `MAC` module should be a simple module and uses the following module parameters:

1. `bufferSize`
2. `maxBackoffs`
3. `backoffDistribution`

As one of its responsibilities the `MAC` module accepts messages of type `AppMessage` from its higher layers and stores them into a local FIFO buffer (provided enough buffer space is available, the number of places in the buffer is given by the `bufferSize` parameter) which for brevity I will call `macBuffer`. When the buffer is full, the `AppMessage` is dropped (and perhaps a logging message is printed). When the buffer was empty before, you trigger the actual MAC process.

The MAC process operates as follows:

1. Extract the oldest `AppMessage` from `macBuffer` and encapsulate it into a message `mmsg` of type `MacMessage`.
2. Initialize a local variable called `backoffCounter` to zero.
3. While `backoffCounter` is smaller than `maxBackoffs` perform the following steps:
   a) Perform a carrier-sense operation.
   b) If the carrier was idle, then transmit `mmsg`, set `backoffCounter` back to zero and go back to the first step.
   c) If the carrier was busy, then increment `backoffCounter` and wait for a random time drawn from the distribution `backoffDistribution`. Then go back to the start of the inner loop.
4. Go back to the first step.

Furthermore, the `MAC` also has a receive path. This is very simple: any packets that arrive from the local `Transceiver` are accepted, the encapsulated application message of type `AppMessage` is decapsulated and sent to the higher layers.

11 Various and Varied Hints

1. I would strongly recommend to use a version control system. In particular, the college of Engineering provides a git server called GitLab at `https://eng-git.canterbury.ac.nz`. You can log in with your UC credentials. Click the plus button in the top righthand corner of the page to create a new project. You can find plenty of information on the web on how to use git.

2. The OMNet++ manual advertises the graphical interfaces as great debugging tools. This should be taken with a grain of salt, in particular the harder errors occur only after some time and are hard to catch. It sounds old school, but **there is no substitute for printing lots of log messages in a structured textual format through which you can sift with `grep`, `awk` and friends.** Do this from the very beginning, write a log message for every interesting event, e.g. arrival of a message, important decisions made during processing etc. Furthermore, make sure that these logs can be easily parsed, e.g. any numbers should not be immediately be followed or preceded by other characters than a blank.

3. You have to be really careful with memory management. In particular, you have to eventually de-allocate (delete) all messages that have been created, otherwise you create memory leaks and you cannot run a simulation for long. A simple guideline is that messages should be deleted by the consumer of the message, for example the `Transceiver` module deletes all incoming `SignalEnd` message after they have been processed, and deletes all `SignalStart` messages once they have been retrieved from the `currentTransmissions` list. For the latter you should have a look at the simulation library of OMNet++.

4. In particular in the `MAC` and the `Transceiver` modules you have to be prepared to receive any message at any time. I suggest to design finite state machines describing the operation of these two modules.

5. You should proceed in small steps and test each step thoroughly before making the next step. In particular, I suggest to proceed in the following order:

   a) First create all the simple module types and the compound module type for a `TransmitterNode` and wire their gates together. These can have empty event handlers initially, the focus is just to get the NED types and the corresponding C++ classes in place and compiled. Do the same for the `ReceiverNode` and the `Channel`.

   b) Make sure that you can create networks with several `TransmitterNode`s, one `ReceiverNode` and a `Channel`, and that the `Channel` is properly connected to all `Transceivers`,

   c) Implement the `PacketGenerator` and add code which just generates, encapsulates, copies and deletes all the required messages but does not show any other behaviour. The goal of this step is to get the message handling and the memory management right.

d) Add the full behaviour to the `Transceiver` module. To test this, create a network with only one `TransmitterNode`, one `ReceiverNode` and the `Channel`. Set up a series of experiments where you vary the distance between the nodes and transmit a number of packets for each distance (see below). Collect statistics about the number of received packets and calculate the packet loss rate from that. What do you expect how the packet loss rate should behave over distance? Do your results match this expectation?

e) Add the full behaviour of the `MAC` module and define suitable scenarios for tests.

6. For the simulation experiments you will be asked to vary a number of system parameters, for example the number M of `TransmitterNode`s. A simulator executable can be supplied with the filename of an `.ini` file as a parameter and I suggest to create separate `.ini` files for the different configurations. To make your life even simpler, I would suggest that you write yourself a small script (perhaps in Python) which **generates** the `.ini` files for you. This way, when you made a mistake, you can correct it in your generator script (and run it again) instead of having to manually edit a number of config files.

Parameter	**Value**	**Comment**
`PacketGenerator.messageSize`	64 bytes	
`Transceiver.txPowerDBm`	0 dBm	corresponds to 1 mW
`Transceiver.bitRate`	250,000 b/s	similar to IEEE 802.15.4
`Transceiver.csThreshDBm`	-50 dBm	
`Transceiver.noisePowerDBm`	-120.0	
`Transceiver.turnaroundTime`	1 ms	
`Transceiver.csTime`	125 μs	
`pathLossExponent`	4	corresponds to indoor propagation
`MAC.bufferSize`	5	
`MAC.maxBackoffs`	5	
`MAC.backoffDistribution`	exponential with 3 ms average	
Max. simulation time	1,000 s	

Table 1: Fixed parameters for all simulation experiments

12 Marking

The marks for the project will be based on the achieved functionality of your code and your simulation results (70%), the final presentation (5%), and the report (25%).

12.1 Simulation Experiments

You are asked to perform a series of simulation experiments. In the following we describe settings that apply to all experiments. Furthermore, in Table 1 the values for several system / module parameters are prescribed.

The single `ReceiverNode` shall always be placed at location $(0, 0)$. If we have M `TransmitterNode`s, these are to be placed equidistantly on the circumference of a circle with given radius R around $(0, 0)$ such that one of these transmitters is placed at location $(R, 0)$. Furthermore, you need to ensure that the `nodeIdentifier` parameters of all nodes are different.

For each fixed set of parameters ten replications are to be carried out, where each replication is started with a different value for the random number seed. The performance metrics for these ten replications shall be averaged and these averages be reported. Each replication shall run for 1,000 simulated seconds.

For the two experiments described below I will ask you to extract certain statistics. It is up to you to figure out where and how to do that.

12.1.1 Channel Packet Loss Rate Simulations

In this experiment we use one `TransmitterNode`, one `ReceiverNode`, and one `Channel`. The radius R (in this case: the distance between transmitter and receiver) will be varied between 1 m and 40 m in steps of 1 m. For the interarrival time distribution `iatDistribution` at the `PacketGenerator` we choose a deterministic distribution with a period of 10 ms. This in effect leads to a fixed number of packets that are being transmitted throughout the 1,000 seconds of simulated time.

For each distance record the number of received packets and record the packet loss rate (defined as the number of received packets divided by the number of transmitted packets). Show the packet loss rate versus distance as a graph. Do not forget to label the axes properly.

12.1.2 MAC Packet Loss Rate Simulations

In this experiment we will vary the number M of `TransmitterNode`s. We fix a radius of $R = 20$ m and vary M as $M \in \{2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$. For the interarrival time distribution `iatDistribution` of the `PacketGenerator` we choose an exponential distribution with a mean of 20 ms.

For these settings to be meaningful and giving the results I expect, make painstakingly sure that all the parameters you use are the ones given in Table 1 and specified here, and furthermore make sure that the BER calculations are correct. In particular, in the case with just one station you should observe a packet loss rate of zero at a distance of $R = 20$ m.

In this experiment we are interested in two different types of "packet losses":

- Packet losses can occur "at the buffer", i.e. a packet generated by a `PacketGenerator` will be dropped by the `MAC` because the buffer is full. I am interested in the percentage of generated packets that are lost "at the buffer".
- The remaining packets can be lost "in the channel", i.e. a `MAC` at a `TransmitterNode` can make an attempt to transmit them but they are not properly received at the `ReceiverNode`. I am interested in the fraction of the remaining packets that get lost "in the channel".

For each number M of `TransmitterNode`s record the packet loss rate at the buffer and the packet loss rate in the channel and show these loss rates versus M as a graph. Do not forget to label the axes properly.

12.2 Final Presentation and Inspection

Each pair is expected to give a final presentation of 10 to 15 minutes duration. This presentation should focus on the following questions:

- What did you find difficult during the development of the simulator and conducting the actual simulation experiments? What were problems / pitfalls and how have you approached them?
- How have you tested your simulator?
- Please show the results for the channel and the MAC packet loss rate simulations, **and explain them**.

Note that you should **not** spend any time discussing the actual implementation or the source code. **Furthermore, please email me the presentation slides in advance**, I would like to have them no later than the evening before presentation.

After the final presentation I will meet with each student individually for about 10 to 15 minutes to assess his/her individual contribution to the project. During this meeting you will explain your source code to me.

12.3 Report

Together with the final presentation each group hands in a report **as a single pdf file by email** to me (please make sure the pdf file has all fonts embedded!). It should contain the following:

- A cover page stating your names and student id's.
- An **agreed-upon** percentage contribution of each partner to the project.
- The figures for the channel and MAC packet loss rate simulations (properly labeled) and your explanation / interpretation of the results.
- A discussion of how you have tested your simulator.
- A discussion of what you found difficult during development / testing / experimenting. What were problems and pitfalls, and how have you approached them?
- It is **not** necessary that you describe the implementation of your simulator in great detail, as the architecture and large parts of the functionality are already given. However, you should **explain your actual MAC algorithm**.
- We have made our life very simple and assumed that collisions always render packets useless and these are being dropped. Can you imagine situations where data might be decodeable despite collisions? And how would you implement that?
- Finally, the source code of your simulator. When possible, please use a pretty

printer.

I would expect that the report comprises between five and ten pages, not including the source code.

References

[1] Bernard Sklar. *Digital Communications – Fundamentals and Applications.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[2] Bjarne Stroustrup. *The C++ Programming Language.* Pearson Education, Upper Saddle River, New Jersey, fourth edition, 2013.