# COSC441-20S2
# Assignment Description – Medium Access Control

Dr.-Ing. Andreas Willig

February 12, 2021

---

## 1 Administrivia

In this assignment you will explore the design, implementation and performance evaluation of two wireless MAC protocols, a CSMA-based protocol and a variant of the venerable ALOHA protocol. The assignment is worth 30% of the overall marks. You will develop a simulation model using the OMNeT++ discrete-event simulation framework. You will be given an overall simulation framework for your MAC protocols, including the "higher layer" components, a transceiver and a channel model. You find that framework on the `Learn` site of the course. It is stored there as a `.tgz` archive, which you will need to unpack it in a suitable place with the command

```
$ tar xvfz file.tgz
```

where `file.tgz` is the filename of the `.tgz` archive. It unpacks into a sub-directory, which contains the file `README.md` with further information.

You will work in groups of two students. The marks for the assignment will be based on:

- The achieved functionality and correctness of your code, and of the results generated by it, as well as the quality of your explanations of the results. You will provide evidence of these in a written final report.
- The quality of presentation of results, as evidenced in the final report.
- Your familiarity with the code and your ability to explain it well, as evidenced in the final report and in one-on-one inspections during week twelve.

I have tried my best to make the following description as clear, helpful and correct/consistent as I possibly can. 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 assignment description as necessary.

### 1.1 Pair Work

One important goal of this assignment is to give you pair-work experience. The rules are as follows:

- You are expected to work in groups of two students. In case of an odd number of students I will allow for at most one submission of a group with three members, who will then have to work on a somewhat expanded project, to be discussed with me.
- Individual submissions will be allocated zero marks.

- It is your responsibility to find a partner.

No exception other than those related to “special consideration” will be given to these rules. Clearly, sometimes things may not work out well with a partner. An important piece of advice in this context is to **start early** with the assignment, so that you have enough time to find a new partner, should that be necessary.

If you do not find a partner on your own (you could try to post on the `Learn` forum!), then you can send me an email. I will then try to find a suitable partner for you from those who sent similar emails.

### 1.2 Plagiarism Warning

Your submissions are logged and originality detection software will be used to compare your solution with other solutions. Dishonest practice, which includes

- letting someone else create all or part of an item of work,
- copying all or part of an item of work from another person with or without modification, and
- allowing someone else to copy all or part of an item of work,

may lead to partial or total loss of marks, no grade being awarded and other serious consequences including notification of the University Proctor.

You are encouraged to discuss the general aspects of a problem with others. However, anything you submit for credit must be entirely your own work and not copied, with or without modification, from any other person. If you need help with specific details relating to your work, or are not sure what you are allowed to do, contact your tutors or lecturer for advice. If you copy someone else’s work or share details of your work with anybody else, you are likely to be in breach of university regulations and the Computer Science and Software Engineering department’s policy. For further information please see

- Academic Integrity Guidance for Staff and Students
  `www.canterbury.ac.nz/ucpolicy/GetPolicy.aspx?file=Academic-Integrity-Guidance-For-Staff-And-Students.pdf`
- Academic Integrity and Breach of Instruction Regulations in the University Calendar
  `www.canterbury.ac.nz/regulations/general-regulations/academic-integrity-and-breach-of-instruction-regulations/`

You will have to sign a plagiarism declaration upon submission of your assignment report.

### 1.3 Assignment Timeline

- During the third term you will acquaint yourself with the C++ programming language and the OMNeT++ discrete-event simulation framework. While the main responsibility for this is on you, I will give a brief introduction to C++, discrete-event simulation and OMNeT++ in the tutorials. You will be able to gain some experience with C++/OMNeT++ with the physical layer assignment.
- Early in term one the assignment description (this document) will be published, and will be updated as necessary. You will be notified through `Learn` about important updates.
- In the second term we will use the tutorials as drop-in clinics.
- At the end of week eleven each pair submits a final report through `Learn`. The precise deadline and submission instructions for the submission are given in Section 8.
- During week twelve I will meet with each of you individually. You will demonstrate your simulation and explain your code to me. You will be marked on your familiarity with the code (**all** of it!) and your ability to explain it clearly and concisely. We will allocate time slots for these inspections closer in time.

# 2 Framework

## 2.1 Module Structure

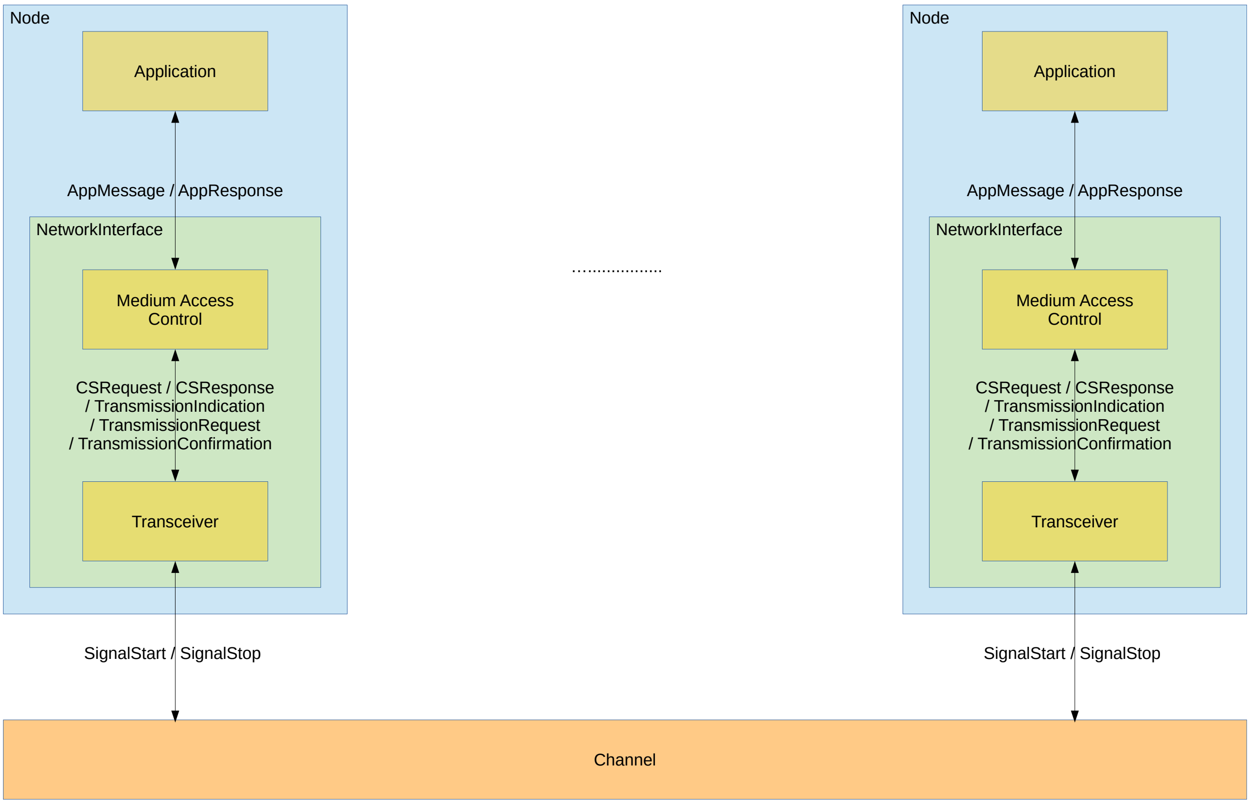


Figure 1: System Setup

The overall module structure of the simulation framework is shown in Figure 1. Note that this does not show the enclosing NED `network` module. All modules mentioned in this subsection are given to you as part of the framework. We discuss the modules in turn.

**Node** A `Node` is an OMNeT++ compound module. It contains as submodules an individual `Application` and a `NetworkInterface` module. It has three parameters: `ownAddress` gives the own node address, and the `ownXPosition` and `ownYPosition` parameters specify the two-dimensional geographic position of the node. These parameters are propagated into the child modules.

**Application** The `Application` is a simple module. It is responsible for generating "application layer" packets and handing these over to the local MAC for transmission, and in the other direction it is responsible for receiving "application messages" from the underlying MAC and generating suitable statistics. It exchanges messages of type `AppMessage` and `AppResponse` with the underlying MAC. It has the following parameters: a configurable distribution for the inter-arrival time of generated messages (parameter `interArrivalTime`), a configurable distribution for the size of the generated messages in bytes (`packetSize`), a configurable distribution for the address of the receiver node of a message (`receiverNodeAddress`), and the own node address (`ownAddress`). The `ownAddress` parameter is set by the enclosing `Node` module, the other parameters should be set in the `omnetpp.ini` configuration file or in the relevant `network` modules for the individual experiments.

**NetworkInterface** The `NetworkInterface` is a compound module containing a MAC module and a `Transceiver` module. The MAC module can be any OMNeT++ module type that adheres to the

`InterfaceMac` module interface, i.e. it is possible to slot in different specific MAC modules (in particular the `CsmaMac` and `AlohaMac` described below). The `NetworkInterface` compound module has four parameters. The three parameters `ownAddress`, `ownXPosition` and `ownYPosition` are set by the enclosing `Node` module and propagated down into the child modules of `NetworkInterface`, the fourth parameter `macType` is a string parameter which you can use to specify the specific MAC module type to be used. This module type will have to adhere to the `InterfaceMac` module interface.

**Transceiver** The `Transceiver` is a simple module, modeling a simplified half-duplex transceiver. It is responsible for the interaction with the `Channel` and provides a simple log-distance channel model. In particular, it keeps track of all ongoing transmissions, calculates for any node the received power of incoming transmissions, calculates bit- and packet error rates and drops incoming packets when they are hit by a bit error or when there has been a collision in the channel (i.e. two or more nodes transmitting at the same time). It can also perform carrier-sense operations for the MAC above, and it models turnaround between transmit and receive mode. Propagation delay is not modeled. The `Transceiver` has a number of parameters:

- `txPowerDBm` is the transmit power in dBm.
- `bitRate` is the bit rate.
- `csThreshDBm` is the carrier-sense threshold in dBm: when the `Transceiver` is tasked to perform a carrier-sense operation, it will calculate the total received power in the channel (from all ongoing transmissions) and compare it against this threshold – if the total power is lower, then the channel/carrier is considered to be free, otherwise busy.
- `noisePowerDBm` is the noise power (in dBm) of the receive circuitry.
- `pathLossExponent` is the path loss exponent.
- `turnaroundTime` is the time (in seconds) the transceiver needs to turn from receive mode to transmit mode (or from transmit to receive mode).
- `csTime` gives the time needed for a carrier-sense operation.
- `ownAddress`, `ownXPosition` and `ownYPosition` are the same as for the `Node` module, and they are set by the surrounding `NetworkInterface`. All other parameters need to be set in the `omnetpp.ini` configuration file.

**Channel** The `Channel` is a simple module responsible for copying messages telling the start and stop of a node transmission to all nodes in the system. These messages are generated and processed by the `Transceiver` modules attached to the channel.

## 2.2 MAC Module Interface

All the modules described so far will be provided to you (both the `.ned` files and the accompanying C++ source code), along with all types of messages and packets that you will need (`.msg` files). You should inspect their `.ned` files for more details on their gate names, parameters and, importantly, their signals and the statistics they generate (if any).

You will have to implement two MAC protocols described below. Both protocols will need to adhere to the module interface `InterfaceMac`, defined as follows:

```
package cosc441_mac;

moduleinterface InterfaceMac
{
    parameters:
```

```
        // own station address
        int     ownAddress;

        // max number of AppMessages the MAC can store in FIFO buffer
        int     bufferSize;

        // Overhead for a MAC data packet
        int     macOverheadSizeData @unit(byte);

        // Overhead for a MAC acknowledgement
        int     macOverheadSizeAck @unit(byte);

    gates:
        input  fromHigher;
        output toHigher;
        input  fromTransceiver;
        output toTransceiver;
}
```

It has the following parameters:

- `ownAddress` is the own station address, set by the enclosing `NetworkInterface` module.
- `bufferSize`: specifies how many `AppMessage` entries for transmission the MAC can hold in total. The buffer is organized in FIFO order, and when it is non-empty, the MAC always works on the oldest (or head-of-line, HOL) element.
- `macOverheadSizeData`: specifies the MAC frame overhead size (MAC header, MAC trailer) in bytes for a MAC data packet.
- `macOverheadSizeAck`: specifies the MAC frame overhead size (MAC header, MAC trailer) for a MAC acknowledgement packet.

Furthermore, a MAC needs to have the following four gates:

- `fromHigher`: This is an input gate over which the MAC will receive packets from the higher layers for transmission. The packets will have to be of type `AppMessage`.
- `toHigher`: This is an output gate, used for two different purposes:
  - The MAC is expected to send back exactly one `AppResponse` message for each `AppMessage` it was asked to transmit by its local higher layers. This `AppResponse` message provides the higher layers with information about the success or failure of packet transmission (see below).
  - Any `AppMessage` received from another node is handed over to the local higher layers.
- `fromTransceiver`: This is an input gate on which the MAC receives messages from the `Transceiver`. The three messages to expect are `CSResponse` (outcome of a carrier-sense operation), `TransmissionConfirmat` (indicating that the `Transceiver` has completed transmission of a packet and is now back in receive mode), and `TransmissionIndication` (indicating that the `Transceiver` has successfully received an incoming packet).
- `toTransceiver`: This is an output gate through which the MAC will send messages to the `Transceiver`. These messages should be of type `CSRequest` (asking the `Transceiver` to initiate a carrier-sense operation) or `TransmissionRequest` (asking the `Transceiver` to start transmission of a packet).

Note that the two `.ned` files `AlohaMac.ned` and `CsmaMac.ned` given to you in the framework conform to this interface.

## 3 Interaction between MAC and `Application`

The `Application` generates packets of type `AppMessage` for transmission and sends these to the MAC, which gets them over the `fromHigher` gate. The precise format of the `AppMessage` packets is shown here:

```
packet AppMessage {
    simtime_t    timestamp;
    int          senderAddress;
    int          receiverAddress;
    int          sequenceNumber;
}
```

Note that class `AppMessage` is a sub-class of the OMNeT++ class `cPacket`, so you can use the `encapsulate()` method of the MAC packet type `MacPacket` to directly encapsulate the `AppMessage` into a MAC packet.

The key requirement in the interaction between MAC and `Application` is that the MAC will have to generate exactly one `AppResponse` message for each `AppRequest` message it has received from the `Application`. The format of the `AppResponse` message is shown here:

```
cplusplus {{
#include "AppMessageStatus_m.h"
}}

enum AppMessageStatus;

message AppResponse {
    int        sequenceNumber;
    int        outcome @enum(AppMessageStatus);
}
```

where the `sequenceNumber` field has to be identical to the `sequenceNumber` field of the `AppMessage` for which the `AppResponse` is generated. There are the following possible values for the `AppMessageStatus`:

```
enum AppMessageStatus
{
    Success          =  0;
    BufferDrop       =  1;
    ChannelFailure   =  2;
};
```

with the following semantics:

- `Success`: means that the MAC protocol has transmitted a `MacPacket` with the `AppMessage` embedded and has received an immediate (MAC layer) acknowledgement for this packet.
- `BufferDrop`: the MAC maintains a finite buffer for `AppMessage`s from the higher layers. If this buffer is full and another `AppMessage` arrives, then the MAC will have to drop the new `AppMessage` (after it generated the `AppResponse` for it, of course).
- `ChannelFailure`: means that the MAC has attempted to transmit a `MacPacket` with the `AppMessage` embedded to the receiver, but has not received an immediate acknowledgement for this packet after exhausting all allowed attempts.[1]

[1] To keep things simple, the MAC protocols you will have to design will not have a broadcast feature. If they had that, then broadcast packets would not be acknowledged and the conditions under which a `ChannelFailure` (or `Success`) are declared would be different for such packets.

# 4 Interaction between MAC and `Transceiver`

The interaction between the MAC and the `Transceiver` can be grouped in three different areas: carrier-sensing for CSMA-type MACs, transmission of packets and reception of packets.

## 4.1 Carrier-Sensing

A CSMA MAC requires help from the `Transceiver` to perform carrier-sensing. The MAC initiates the process by sending a `CSRequest` message to the `Transceiver` (through the `toTransceiver` gate). This message carries no parameters and is declared as follows:

```
message CSRequest {
}
```

After receiving this message the `Transceiver` will spend some time to carry out the carrier-sense operation, and then the `Transceiver` will return a `CSResponse` message, declared as follows:

```
message CSResponse {
    bool      busyChannel;
}
```

If `busyChannel` is `true`, then the channel is considered busy (i.e. the total power received on the channel exceeds the carrier-sense threshold `csThreshDBm`), otherwise the channel is considered idle. After receiving an idle `CSResponse` a CSMA MAC will proceed to initiate the packet transmission process.

## 4.2 Packet Transmission

To prepare a packet transmission, the MAC first prepares a `MacPacket`, defined as follows:

```
cplusplus {{
#include "MacPacketType_m.h"
}}

enum MacPacketType;

packet MacPacket
{
    int     receiverAddress;
    int     transmitterAddress;
    int     macPacketType @enum(MacPacketType);
}
```

where `MacPacketType` is defined as

```
enum MacPacketType
{
    MacDataPacket   =  0;
    MacAckPacket    =  1;
};
```

More precisely, after creating a new instance of `MacPacket`, the MAC will have to fill in fields of the `MacPacket`:

- The `receiverAddress` is to be taken from the `AppMessage` to be transmitted.
- The `transmitterAddress` is filled with the value of the `ownAddress` parameter of the MAC.

- The `MacPacketType` tells whether the packet is a data packet or an ACK packet.

Besides filling in these fields, the MAC will also have to set the length of the `MacPacket` (yet without payload), using the `setByteLength()` method. Depending on the `MacPacketType`, this length will either be the value of the `macOverheadSizeData` parameter or of the `macOverheadSizeAck` parameter. After this, and when the packet is of type `MacDataPacket`, the `AppMessage` to be transmitted can be encapsulated into the `MacPacket` using the `encapsulate()` method.

The resulting `MacPacket` is then encapsulated (using `encapsulate()` again) into a new message of type `TransmissionRequest`, defined as follows:

```
packet TransmissionRequest {
}
```

The resulting `TransmissionRequest` message is then sent to the `Transceiver` via the `toTransceiver` output gate.

After some activity (the `Transceiver` will first turn around from receive mode to transmit mode, then transmit the actual packet over the channel and finally turn back from transmit to receive mode), the `Transceiver` will send back a `TransmissionConfirmation` message to the MAC, which has the following format:

```
message TransmissionConfirmation {
}
```

This message will arrive at the MAC on the `fromTransceiver` gate. It means that the packet transmission has been completed and the `Transceiver` is now back in receive mode.

### 4.3 Packet Reception

Most of the hard work in packet reception is carried out in the `Transceiver` module, in particular the evaluation of the channel error model and figuring out whether an incoming packet has bit errors or not. If an incoming packet is free of bit errors and has a destination address matching the `ownAddress` parameter of the `Transceiver`[2] (set by the enclosing `NetworkInterface` module to the same value as the MAC module parameter of the same name), then the `Transceiver` will send a `TransmissionIndication` message to the MAC, which it will receive on its `fromTransceiver` gate and which has the following format:

```
packet TransmissionIndication {
}
```

The actual MAC packet of type `MacPacket` is encapsulated into this message and can be decapsulated using the – surprise! – method `decapsulate()`. The MAC will then have to retrieve the `AppMessage` by `decapsulate()`-ing it out of the `MacPacket` and send it to the `Application` through the `toHigher` gate.

## 5 The CSMA MAC Protocol

The first, and more complex MAC protocol that you should design and implement is a CSMA-type protocol with immediate acknowledgements, a configurable number of transmission attempts (initial attempt and some retransmissions) and a configurable number of random carrier-sense backoffs per attempt.

Your CSMA MAC protocol will be a simple OMNeT++ module called `CsmaMac`. A minimal `.ned` specification is provided to you and looks as follows:

[2] Recall that we do not support broadcast/multicast, so incoming packets are **only** accepted when their destination address matches the node address.

```
package cosc441_mac;

import cosc441_mac.InterfaceMac;

simple CsmaMac like InterfaceMac
{
    parameters:
        @signal[bufferLossSig](type=bool);          // number of packets lost at MAC buffer
        @signal[bufferEnteredSig](type=bool);       // number of packets admitted to MAC buffer (and transmission)
        @signal[numberAttemptsSig](type=long);      // number of attempts per packet
        @signal[accessFailedSig](type=bool);        // number of packets for which no ACK is ever received
        @signal[accessSuccessSig](type=bool);       // number of packets for which an ACK is received
        @statistic[bufferLossSig](record=count; title="MAC: number of higher-layer packets lost at buffer");
        @statistic[bufferEnteredSig](record=count; title="MAC: number of higher-layer packets admitted to buffer");
        @statistic[numberAttemptsSig](record=stats; title="MAC: number of attempts per packet");
        @statistic[accessFailedSig](record=count; title="MAC: number of MAC packets for which no ACK was received");
        @statistic[accessSuccessSig](record=count; title="MAC: number of MAC packets for which an ACK was received");

                 int      ownAddress;                              // own node address
                 int      bufferSize;                              // max number of application messages the MAC can hold
                 int      maxBackoffs;                             // max number of backoff operations per attempt
                 int      maxAttempts;                             // max number of attempts
                 int      macOverheadSizeData @unit(byte);         // Overhead for a MAC data packet
                 int      macOverheadSizeAck @unit(byte);          // Overhead for a MAC acknowledgement
                 double   macAckDelay @unit(s);                    // fixed waiting time for sending ACK
                 double   ackTimeout @unit(s);                     // timeout for ACK packet
        volatile double   csBackoffDistribution @unit(s)     = default(uniform(0ms, 3ms));     // backoff after busy CS
        volatile double   attBackoffDistribution @unit(s)    = default(uniform(0ms, 10ms));    // backoff after failed attempt
        volatile double   succBackoffDistribution @unit(s)   = default(uniform(0ms, 3ms));     // backoff after successful atte

    gates:
        input    fromHigher;
        output   toHigher;
        input    fromTransceiver;
        output   toTransceiver;
}
```

In the following the operation of the CSMA MAC is only described in broad strokes. You will need to design (and implement!) its detailed operation.

## 5.1 Packet Buffer

An important design element of the MAC is that it only holds a finite number of packets to transmit. This number is given by the value of the `bufferSize` module parameter, which has to be an integer larger than or equal to one.

The buffer is organized in FIFO order. The MAC always works on the oldest (or head-of-line, HOL) packet and tries to transmit it to the receiver. When the outcome for the HOL packet is known (i.e. the MAC has either received an acknowledgement for it or has exhausted all allowed attempts), it is removed from the buffer and an appropriate `AppResponse` message describing the fate of this packet is sent to the `Application` through the `toHigher` gate. After this, the MAC will check the buffer status again and starts to work on the next packet when the buffer is non-empty.

When a new `AppMessage` arrives from the higher layers (through the `fromHigher` gate), then the buffer occupancy is checked. If there are currently strictly less than `bufferSize` packets in the buffer (not counting the newly arrived packet) then the packet is added to the buffer (becoming the tail element of the queue) and, if it is the only packet in the buffer, the MAC shall start to work on this packet immediately. If the buffer occupancy upon arrival of a new packet is `bufferSize`, then the buffer is full, and the newly arrived `AppMessage` is to be discarded after the MAC generated an appropriate `AppResponse` message reporting a buffer loss.

## 5.2 Receive Path and Acknowledgement Generation

When the `Transceiver` receives a packet without bit errors and if the destination address of the packet coincides with `ownAddress`, then it will hand over the packet to the MAC using a `TransmissionIndication`, into which the `MacPacket` is encapsulated. After receiving the `TransmissionIndication`, the contained `MacPacket` should be decapsulated. The `AppMessage` encapsulated within the `MacPacket` should be decapsulated as well and sent to the higher layers (through the `toHigher` gate).[3]

[3] The rule that the MAC hands over *every* data packet it receives to the higher layers leads to a minor problem: the MAC may hand over the same packet more than one time, which can happen when the original sender of the packet does not receive an acknowledgement properly and re-transmits the same packet again. In this assignment the MAC does not

The MAC is required to respond to a received unicast `MacPacket` with an immediate acknowledgement, to be generated after a fixed amount of time, given by the `macAckDelay` module parameter.[4] Hence, after getting the `TransmissionIndication` the MAC will have to wait for time `macAckDelay`, and when this time has expired (you should use a self-message for this timeout) then the MAC shall prepare a `MacPacket` of type `MacAckPacket` and hand it over to the `Transceiver` encapsulated into a `TransmissionRequest` message. No carrier-sense operation is to be carried out before sending the ACK packet.

### 5.3 Transmit Path

The MAC always works on the head-of-line (HOL) packet in the buffer. We first describe what constitutes an *attempt* and what the different outcomes of an attempt are:

- An attempt always starts with a carrier-sense operation (i.e. exchanging `CSRequest` and `CSResponse` messages with the `Transceiver`).
- If the `Transceiver` indicates an idle carrier, then the MAC shall send a packet, by carrying out the following steps:
  - Create a `MacPacket` of type `MacDataPacket` and encapsulate the HOL `AppMessage` into it.
  - Create a `TransmissionRequest` message, encapsulate the `MacPacket` and send it to the `Transceiver`.
  - Wait for the `Transceiver` returning back a `TransmissionConfirmation` message.
  - After getting the `TransmissionConfirmation` message, start a timer (we call it the ack timer here) with the timeout being `ackTimeout` seconds into the future.
  - If the MAC receives an acknowledgement packet before the ack timer expires, then the packet transmission has been successful (and the local `Application` is informed by sending it an `AppResponse` message indicating `Success`), the ack timer is cancelled, the current attempt ends (and is considered successful) and the current HOL packet in the buffer is dropped (the packet has been successfully completed). After this, the MAC will wait an additional random backoff time, which here we refer to as a *success backoff*,[5] and which is drawn from the `succBackoffDistribution` module parameter. After this backoff waiting time, the contents of the packet buffer is checked. If the buffer is non-empty, the MAC will start working on a new packet.
  - If the MAC does not receive an acknowledgement before the ack timer expires, then upon expiry of the ACK timer the current attempt shall be deemed failed.
- If the `Transceiver` indicates a busy carrier, the MAC follows a non-persistent-CSMA-type of strategy by executing a *carrier-sense backoff*: the MAC will draw a random waiting time from the `csBackoffDistribution` module parameter, wait for this time, and trigger the next carrier-sense operation.
- The number of carrier-sense backoff operations per attempt is limited to the value given in the `maxBackoffs` module parameter. If the carrier-sense operation after the `maxBackoffs`-th carrier-sense backoff again indicates a busy medium, then the attempt shall be deemed failed. It is important to note that different attempts shall each have the full number `maxBackoffs` of carrier-sense backoffs.

When an attempt is deemed as failed, the MAC will wait for a random backoff time drawn from the `attBackoffDistribution` given as a module parameter – we refer to this backoff as a *failure backoff*.

need to take care of this, instead duplicates are identified in the `Application` module, which generates a suitable signal / statistic.

[4] In this assignment the MAC implementation generates the acknowledgement, which is the correct option from a layering perspective. In practice it is common for ACKs to be generated in the physical layer by the Transceiver chip.

[5] The intention of this additional backoff is for the successful station to step back for a while and give other stations a chance.

Afterwards the MAC will check whether further attempts are allowed, the total number of allowed attempts is given by the `maxAttempts` module parameter.

### 5.4 Signals/Statistics

The MAC should collect a number of statistics, using OMNeT++ mechanism for signal-based statistics collection (see Sections 4.14 and 4.15 in the OMNeT++ simulation manual ). You should collect at least the following statistics:

- Number of `AppMessage`'s arriving from the local `Application` that are dropped at the buffer (i.e. `AppMessage`'s arriving at a full buffer that then get dropped).
- Number of `AppMessage`'s arriving from the local `Application` that are admitted to the buffer (i.e. not dropped).
- The number of attempts per `AppMessage`.
- The number of `AppMessage`'s for which no acknowledgement is ever received.
- The number of `AppMessage`'s for which an acknowledgement is received.

These statistics can help your debugging. They are pre-defined in the skeleton file `CsmaMac.ned` given to you as part of the framework.

## 6 The ALOHA MAC Protocol

Once you have implemented the `CsmaMac` module, it is an easy exercise to modify it and implement an ALOHA-type protocol: in the CSMA MAC protocol described in Section 5 the carrier-sense operation does not actually need to be carried out, but is considered successful right from the start (and no time is spent on carrier-sensing).

The ALOHA MAC should become its own OMNeT++ simple module `AlohaMac` with the same module parameters and gates as `CsmaMac`. It's implementation should be sub-classed from the `CsmaMac` implementation. A skeleton implementation of the `AlohaMac` module is provided in the framework.

## 7 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 and 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 sprinkling your code generously with assertions and printing lots of log messages in a structured textual format through which you can easily sift with `grep`, `awk` and friends.** Do this from the very beginning, write a log message (using OMNeT++'s standard mechanism of sending text to the `EV` output stream) for every interesting event, e.g. arrival of a message, timer events, important decisions made during processing etc. I suggest that you inspect the source code of the `Transceiver` for some inspiration and some useful helper methods to implement. OMNeT++ allows (through a configuration setting) to store these messages in a log file. Make sure that these logs can be easily parsed, e.g. any numbers should not 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 deallocate (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, when the `Transceiver` sends a `TransmissionConfirmation`, then the MAC should delete that message.

4. When designing and developing the `CsmaMac`, you should proceed in small steps and test each step thoroughly before making the next step. For example:

   a) Start with a simple CSMA MAC that performs carrier-sensing, transmission of data packets and acknowledgements, but does not yet implement carrier-sense backoffs and multiple attempts. Test this thoroughly in a simple scenario with just one "transmitter node" and one "receiver node" that are very close to each other (to rule out channel errors). More precisely, the `receiverNodeAddress` parameter of the "transmitter node" should point to the "receiver node", and it should generate new packets of a fixed length deterministically every ten seconds or so (`interArrivalTime` parameter), whereas the `interArrivalTime` parameter of the "receiver node" should be set to a very huge deterministic value much larger than your simulation stopping time (so it never generates a packet). Inspect your log messages to make sure all steps are taken in the right order, that packet transmissions have the duration you expect, that timeouts (like the `ackTimeout`) work correctly, and so on.

   b) Next set the distance between the transmitter node and receiver node to a larger value where packet/bit errors occur at a high rate, and check whether your transmitter node notices the absence of an acknowledgement in such a case.

   c) Next implement having multiple attempts and test in the previous two-node scenario whether it works correctly.

   d) Next test whether you have implemented buffer handling correctly by vastly increasing the rate of `AppMessage` arrivals at the transmitter node to the point where packets arrive faster than they can be transmitted.

   e) Next implement carrier-sense backoffs and test them in a scenario with multiple transmitter nodes that are close to each other and to the receiver.

   f) And so on ...

5. Write yourself a separate script that sifts through the results (stored in `.sca` files) and generates the plots, e.g. using Python and matplotlib or – if you want to do it tough – using `gnuplot`. I cannot over-emphasize how time-saving it is to use scripts for this instead of manually creating figures in, say, Excel. Chances are you will have to do this multiple times.

6. **Beware:** You should plan enough time for the actual production simulations! This can be in the order of days!

There is one important guideline for designing the (implementation of) the `CsmaMac` module. I strongly suggest you design it as an extended finite state machine, where you identify a number of states and identify a set of events (e.g. packet / message arrivals, timeouts) that can happen. A *state* should signify a step in the operation of the MAC where the MAC actually waits for something to happen before it can proceed – e.g. it waits for a `CSResponse` after having sent a `CSRequest`, it waits for an acknowledgement packet or an ack timeout after having sent a data packet, it waits for the expiry of a backoff time (either a carrier-sense backoff, a success backoff or a failure backoff – these are three distinct events), it waits for the expiry of the fixed waiting time before sending an acknowledgement itself, etc. After you have identified the states, you will next need to identify all the possible events that can happen (i.e. all the possible messages that the MAC can receive), including the various timeout events, the arrival of an `AppMessage`, the arrival of a `TransmissionIndication`, and so on. Once you have identified all the states and all possible events, you will need to specify **for each event and each state** how the event should be handled in that particular state. If a particular combination is nonsensical, then you should stop your simulation with an error. An example of such a situation is when you receive a `CSResponse` message while you are waiting for an acknowledgement for a data packet that you have just finished sending – it is clear from the protocol description that this should not happen. In other cases you will have to make actual design decisions. For example, what happens if your MAC receives a data packet addressed to itself (for which it should send an acknowledgement) while it waits itself for an acknowledgement for a previous data packet. Should it send an acknowledgement itself and abort its own data transmission procedure or should it not send an acknowledgement and hope to get one itself?

# 8 Deliverables and Assessment

## 8.1 Marking Scheme

Marking will be based on two artefacts:

- A final report.
- A one-on-one inspection.

The marks you get for the assignment are determined in three steps. First I determine *objective marks*, then I determine *individual marks*, which then may get adjusted due to different percentage contribution.

In the first step I will determine the *objective marks*, which are the same for both partners in a pair and which are based on:

- The completeness of the implementation and its correctness (e.g. whether all relevant states and operating conditions are properly handled, amount of error / fault detection and handling, etc.). This amounts to 40% of the objective marks.
- The correctness of the simulation results and the quality of your explanations for these. This amounts to 30% of the objective marks.
- The quality of your explanation of the MAC protocol implementation, of your explanation of testing, and other aspects of the report (including the general writing). This amounts to 30% of the objective marks.
- I may apply deductions to these objective marks, e.g. based on the quality / readability of your source code or glaring implementation flaws not otherwise covered. The deductions may be up to 15% of the achievable objective marks.

These objective marks count for 90% of the individual marks. The remaining 10% of the individual marks are awarded after the one-on-one inspections, where I will assess your familiarity with the code and your ability to explain it clearly and concisely to me. To get full marks you have to be able to explain *everything*! Should your explanations be so weak that I get the impression that you have contributed much less to the code than you claim you have, I reserve the right to apply much more serious deductions than just 10%.

In a final step these individual marks are adjusted according to the different percentage contribution of each partner in a pair. The partner doing more will get marks added to the individual marks, the other partner will get marks deducted. However, this will not happen in a balanced fashion, the better-contributing partner gets fewer extra marks than the other partner loses.

## 8.2 Inspections

I will meet with each student individually for about 20-25 minutes. During the first ten minutes you demonstrate to me your simulation, i.e. you start it, run it and show me how you generate results. The second ten to fifteen minutes will be a code-walk-through, where you explain the source code to me. You will be assessed on your knowledge of the source code and your ability to explain it clearly and concisely. The time slots for these inspections are arranged closer in time.

## 8.3 The Final Report

Each pair hands in a report **as a single pdf file** through `Learn`. It should contain the following:

- A cover page stating your names and student id's, a word count for the body of the report, an **agreed-upon** percentage contribution of each partner to the assignment, and for each partner a very brief summary of the main contributions to the assignment (so that I get an idea how you have split up the work).

- The body of the report should have the following chapters:
  - MAC design and implementation: Here you give an account of your CsmaMac design and implementation in good detail. In particular, describe the states and the reaction to each and every event in each and every state. Please also briefly explain how the AlohaMac differs.
  - Testing: Explain how you have tested the CsmaMac. Describe for each test what you wanted to test, how you have tested it and what the results were.
  - Reflection: What have you designed / implemented / tested well, and what can be improved.
  - Results: Please show your simulation results. The required experiments and the results that you are asked to show are described in Section 8.4. In your results indicate the achieved statistical confidence.
  - Explanation of results: You are asked to provide an explanation for your results (see Section 8.4). I want to know *why* your results look the way they do. I am not interested in an explanation of *what* they look like.
  - Improvements: Please explain how your CsmaMac protocol implementation would need to be modified to implement broadcast/multicast packet handling and a binary exponential carrier-sense backoff. Furthermore, please explain how the problem of duplicates mentioned in Footnote 3 can be solved on the MAC layer instead of solving it in the Application module. Do you have any other suggestions for improving the CsmaMac protocol?
- The source code of your CsmaMac and AlohaMac modules (including all .ned, .h, and .cc files for these modules), your network definitions (.ned files) for all experiments specified in Section 8.4, as well as your omnetpp.ini file, should be included in an appendix to your report.
- You have to print the plagiarism declaration form from the Learn page of COSC 441, sign it, scan it, convert the scanned form into a pdf and include this into your report as well.
- Please also send me the source code for CsmaMac and AlohaMac and your omnetpp.ini by email.

I would expect that the report comprises between ten and fifteen pages (between 5,000 and 7,500 words), not including the source code, the cover page and the plagiarism declaration.

- **Submission deadline**: Friday, May 28, 2021, 11:59pm.
- The report is to be submitted on Learn.

## 8.4 Simulation Experiments

In this section I describe the simulation experiments that need to be performed.

### 8.4.1 Setup and Common Parameters

We first describe settings that apply to all experiments, these are given in Table 1. Many of them are identical to their corresponding values in the physical layer assignment. Note that the framework already contains a skeleton omnetpp.ini file in which these parameters are set to the values given here.

For each fixed set of parameters (a *measurement*) you should carry out a number of repeat independent replications, and the key performance indicators for each experiment (specified below) obtained from all the replications are to be averaged. You are then asked to report the achieved confidence intervals for confidence levels of 95% and 99%. Each replication shall run for 1,000 simulated seconds. You can report the confidence intervals either in a table or in the graphs (using error bars).

| Parameter | Value | Comment |
|---|---|---|
| `sim-time-limit` | 1000 s | |
| `repeat` | 30 | this many replications per measurement |
| `Application.packetSize` | deterministic, 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` | 300 $\mu$s | |
| `Transceiver.csTime` | 125 $\mu$s | |
| `Transceiver.pathLossExponent` | 4 | corresponds to indoor propagation |
| `MAC.bufferSize` | 5 | |
| `MAC.maxBackoffs` | 5 | |
| `MAC.maxAttempts` | 3 | |
| `MAC.macOverheadSizeData` | 20 byte | |
| `MAC.macOverheadSizeAck` | 20 byte | |
| `MAC.macAckDelay` | 500 $\mu$s | |
| `MAC.ackTimeout` | 1.5 ms | |
| `MAC.csBackoffDistribution` | uniform from 0 ms to 6 ms | |
| `MAC.attBackoffDistribution` | uniform from 0 ms to 6 ms | |
| `MAC.succBackoffDistribution` | uniform from 0 ms to 12 ms | |

Table 1: Fixed parameters for all simulation experiments

In the following I am going to use the terms “transmitter” and “receiver” nodes. A “receiver node” is a node where the `interArrivalTime` parameter of the `Application` is set to a deterministic value much larger than the `sim-time-limit`, so that it really never sends a packet. The setting of the `packetSize` and `receiverNodeAddress` parameters of the receiver does not matter. A “transmitter node” is configured with a given `packetSize`, an `interArrivalTime` distribution with an average inter-arrival time much smaller than `sim-time-limit` (so that it effectively will send packets) and a `receiverNodeAddress` that is pointing towards a receiver node.

The assignment framework contains an OMNeT++ compound module called `CircularNetwork` (in file `CircularNetwork.ned`) which places one “receiver node” in the origin of the two-dimensional plane, and a configurable number of transmitter nodes equidistantly on a circle of configurable radius around the origin. This module can (and should!) be used when specifying the OMNeT++ networks for all the experiments below.

### 8.4.2 Experiment One

In the first experiment we have just one transmitter node and one receiver node, placed at a distance of 5m from each other (so that there should be practically no packet losses in the channel due to bit errors). Use only the `CsmaMac` protocol. We consider two different cases for the `interArrivalTime` parameter of the `Application`:

- Deterministic inter-arrival times from the set $\{1\text{ms}, 2\text{ms}, 3\text{ms}, \ldots, 20\text{ms}\}$
- Exponentially-distributed inter-arrival times with average inter-arrival times from the same set.

In addition, for this particular experiment please set the parameter `MAC.succBackoffDistribution` to the deterministic value of zero.

For each `interArrivalTime` parameter value collect the number of packet losses at the buffer (of the transmitter node) and the number of successfully received non-duplicate packets at the receiver node (it has a dedicated statistic for that, called `rxUniquePacketsSig`, see `Application.ned`) and display them graphically (with the (average) inter-arrival time on the $x$-axis), with one curve for deterministic and one curve for exponential inter-arrival times. Use a logarithmic scale on the $y$-axis – in doing so, particularly for the number of dropped packets I suggest you add a 1 to each outcome. Please explain your findings.

### 8.4.3 Experiment Two

We have one receiver node and ten transmitter nodes placed equidistantly on a circle of radius $R \in \{2\text{m}, 3\text{m}, 4\text{m}, \ldots, 20\text{m}\}$, with the receiver node being at the center of the circle. Each transmitter node generates packets with an exponential inter-arrival time distribution with an average of 15 ms. Do this for both the ALOHA and the CSMA protocols. Record the number of successfully received non-duplicate packets at the receiver node and show it graphically (one curve per protocol, logarithmic scale on the $y$-axis), with the radius $R$ on the $x$-axis. Please explain your findings.

### 8.4.4 Experiment Three

We have one receiver node and a varying number of transmitter nodes, all placed equidistantly on a circle of radius 5m around the receiver node. The number of transmitter nodes is to be varied in $\{2, 3, 4, \ldots, 20\}$. At each transmitter packets arrive with an exponentially distributed inter-arrival time of 15 ms average. Do this for both ALOHA and CSMA. Record the number of successfully received non-duplicate packets at the receiver node and show it graphically (one curve per protocol, logarithmic scale on the $y$-axis) with the number of transmitters on the $x$ axis.

### 8.4.5 Experiment Four

Same as experiment three, but now with a radius of 20m around the receiver node.