Computer Networks Lab introduced for ‘Computernetwerken’

by Andrei Belogaev

Name 1
Name 2
Group groupID

May 25, 2023

Introduction

Before diving into this computer networks lab course, it is useful to present some conventions first. ***Keep in mind that these conventions are introduced to make all of our lives easier.*** In case of problems, it is easier to have a look at a lab setup if everyone sticks to the same rules.

6.1 Practical Arrangements

Part 1. Virtual Machine with NS-3

Getting the VM

We provide a Virtual Machine with an Ubuntu variant and NS-3 installed. You can download this image from `https://student.idlab.uantwerpen.be/computernetwerken/`. There are two distinct VMs available: one for running on Intel-based computers (“Computernetwerken-AMD64”), and one for running on Apple silicon (“Computernetwerken-ARM”).

If you are using a computer running on Intel architecture, we suggest you use Virtualbox (`https://www.virtualbox.org/`) for running your Virtual Machine. After installing Virtualbox, set up your machine as follows:

1. Download and open the Computernetwerken-AMD64.ova file.
2. You should now see the VirtualBox Import Appliance.
3. Click the Import button, and the Virtual Machine is imported in Virtualbox. This can take a few minutes.

When you are using a machine running on Apple silicon, A VM is provided that has been made with the “UTM” virtualisation software. UTM is freely downloadable from `https://getutm.app`. If your are using Apple silicon, set up your machine as follows after installing the UTM appliance:

1. Download and open the Computernetwerken-ARM.utm.zip file.
2. Unpack the zip file.
3. You should now see the UTM Appliance.
4. Double-click the Computernetwerken-ARM.utm file to import it into UTM.

Now you should be able to start your virtual machine. The login is `computernetwerken`, and the password is `mvkbj1n` (from the Dutch phrase *“Met veel kabels bouw je 1 netwerk”*).

Working with Git

For version control maintenance, the NS-3 project relies on *Git* (https://git-scm.com/book/), a powerful and widely used distributed version control system. Version control is essential for tracking changes in files over time, allowing easy retrieval of specific versions later on. When working with Git, each file in your working directory can be classified as either *tracked* or *untracked*. Tracked files are those that Git monitors and maintains a history of their modifications, while untracked files are not managed by Git.

One of the key concepts in Git is the notion of a *revision*. A revision represents the state of the entire project at a specific point in time and includes all the tracked files within the project. To view the revision history of the NS-3 project or any Git repository, you can navigate to the root directory of the project `~/ns3-student`, and execute the following command:

```
git log
```

Running this command will display a list of *commits*, each representing a specific revision of the project. The commit history includes information such as the commit hash (a unique identifier), the author, the date and time of the commit, and an associated commit message that describes the changes made in that revision. Using the commit hash, you can easily revert the whole project to a certain version with command

```
git checkout <commit_hash>
```

If we modify a tracked file, Git changes the status of this file to *modified*. As an example, you can add your name into the list of authors `~/ns3-student/AUTHORS`. After that, check the status of the project with the following command:

```
git status
```

This command will show you that the file AUTHORS is modified. To see the difference between the project's current version and the one corresponding to the last commit, you can use the command

```
git diff
```

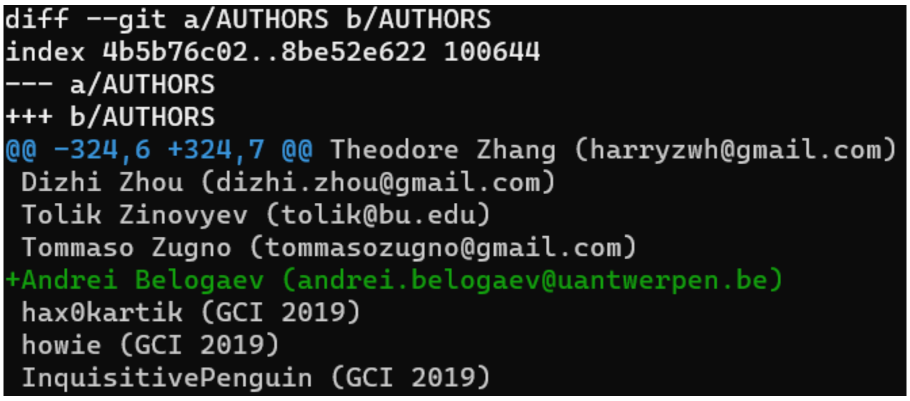
Running this command from the project root directory will show you the differences for all the tracked files. Since we modified only one file, we see the information about this file (cf., Fig. 6.1):

To see the difference for a particular file, you can run:

```
git diff AUTHORS
```

It might happen, that you need the original version of the file, but you do not want to loose the changes that you made. Then, you can "hide" the changes with the command

```
git stash -- AUTHORS
```



```
diff --git a/AUTHORS b/AUTHORS
index 4b5b76c02..8be52e622 100644
--- a/AUTHORS
+++ b/AUTHORS
@@ -324,6 +324,7 @@ Theodore Zhang (harryzwh@gmail.com)
 Dizhi Zhou (dizhi.zhou@gmail.com)
 Tolik Zinovyev (tolik@bu.edu)
 Tommaso Zugno (tommasozugno@gmail.com)
+Andrei Belogaev (andrei.belogaev@uantwerpen.be)
 hax0kartik (GCI 2019)
 howie (GCI 2019)
 InquisitivePenguin (GCI 2019)
```

Figure 6.1: Git diff example.

If you run again the commands diff and status, you will see that there are no changes in the project. However, these changes are not lost – they are safely stored by Git. Moreover, you can have multiple changes that are stored in a stash list. To see the entries of the list, you can run

```
git stash list
```

After executing this command, you will see the only entry in the list, with label stash@{0}. To see the content of this stash, you can run

```
git stash show -p stash@{0}
```

To reapply the changes, run

```
git stash apply stash@{0}
```

If you run the commands diff and status, you will see that the file AUTHORS is again modified. Note that the entry is not deleted from the stash list. To delete this entry, run

```
git stash drop stash@{0}
```

Sometimes you need to completely revert the changes that you have made in a particular file. To return the file AUTHORS to its original version corresponding to the latest commit, you can run:

```
git checkout -- AUTHORS
```

To completely revert all changes in all tracked files, you can run from the project root directory the following command:

```
git checkout .
```

Be careful with the last command! After executing it, you lose all the changes that you have made.

6.2 NS-3: Introduction

In this lab, you will acquaint yourself with the NS-3 simulation platform. Using this model, you will simulate two medium access control (MAC) layer protocols, Aloha and DCF, which specify how multiple devices of the same wireless network can access the medium. The modification of the latter one is used in modern wireless local area networks that we all know under the name Wi-Fi.

Part 1. Basics of NS-3

NS-3 is a discrete-event network simulator, targeted primarily for research and education use. This is a free open source software written in the C++ programming language. Using NS-3, you can conduct experimental studies on different communication technologies and network topologies. It contains implementations of protocols that are parts of many commonly used telecommunication technologies, for example, Wi-Fi and LTE/5G. The NS-3 repository can be accessed at `https://gitlab.com/nsnam/ns-3-dev/`, all the information and documentation can be found at `https://www.nsnam.org/`.

It is important to understand the difference between emulation and simulation: Mininet is a network emulator, while NS-3 is a simulator. Emulators work in real time and precisely mimic the behavior of real hardware and software in a real environment. In contrast, simulators maintain their own simulation clock and do not precisely follow all the procedures that occur in real devices and software. However, they approximate the environment and the devices behavior in an efficient way so that they provide a good accuracy of the experiment outcomes. For example, NS-3 does not emulate all operation system (e.g., Linux) procedures. It also does not emulate the signal generation and transmission over the wireless channel, but uses mathematical equations to estimate the error probability. Another very important feature provided by simulators is the experiment repeatability. It means that two or more runs of the simulation model with the same parameters will produce precisely the same results. Any randomness in a simulation model occurs due to the use of pseudorandom numbers to generate a certain event (see implementation details at `https://www.nsnam.org/docs/manual/html/random-variables.html`), e.g., error during data transmission event with a known probability obtained from an equation.

Exercise 1: Building and running NS-3

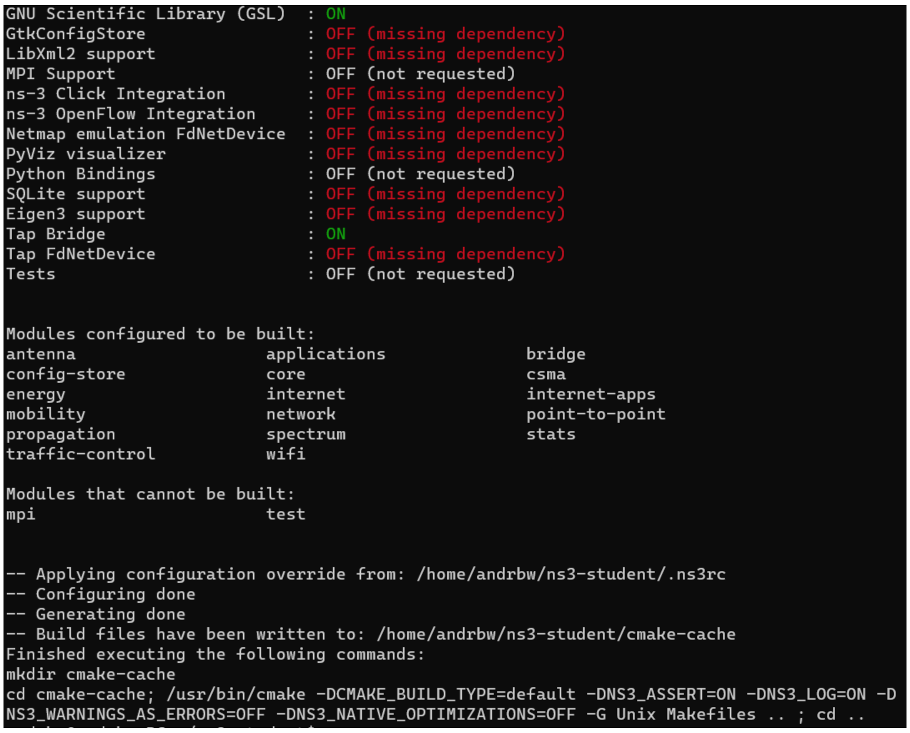
This exercise walks you through the steps of building NS-3 and running an experiment (see also NS-3 tutorial `https://www.nsnam.org/docs/tutorial/html/`). The exercise is conducted on the virtual machine.

1. NS-3 uses the Cmake system to build the project. To make the configuration and building easier for users, it provides a Python wrapper around Cmake, called ns-3, that symplifies the command-line syntax. You can find this wrapper in the root directory of the project, in our case it is `~/ns3-student`. There are several options to control the build, e.g., enabling tests and examples, debug mode, etc. We will use the default build configuration. This is done by executing the following command:

   ```
   % ./ns3 configure
   ```

   When the command finishes execution, you should see the output in command line similar to the one on Fig. 6.2.

2. Now we can build the NS-3 project. In directory `~/ns3-student/src` we can see many subdirectories, each of which contains source files for a particular module of NS-3. Since



```
GNU Scientific Library (GSL)  : ON
GtkConfigStore                : OFF (missing dependency)
LibXml2 support               : OFF (missing dependency)
MPI Support                   : OFF (not requested)
ns-3 Click Integration        : OFF (missing dependency)
ns-3 OpenFlow Integration     : OFF (missing dependency)
Netmap emulation FdNetDevice  : OFF (missing dependency)
PyViz visualizer              : OFF (missing dependency)
Python Bindings               : OFF (not requested)
SQLite support                : OFF (missing dependency)
Eigen3 support                : OFF (missing dependency)
Tap Bridge                    : ON
Tap FdNetDevice               : OFF (missing dependency)
Tests                         : OFF (not requested)


Modules configured to be built:
antenna                   applications              bridge
config-store              core                      csma
energy                    internet                  internet-apps
mobility                  network                   point-to-point
propagation               spectrum                  stats
traffic-control           wifi

Modules that cannot be built:
mpi                       test


-- Applying configuration override from: /home/andrbw/ns3-student/.ns3rc
-- Configuring done
-- Generating done
-- Build files have been written to: /home/andrbw/ns3-student/cmake-cache
Finished executing the following commands:
mkdir cmake-cache
cd cmake-cache; /usr/bin/cmake -DCMAKE_BUILD_TYPE=default -DNS3_ASSERT=ON -DNS3_LOG=ON -D
NS3_WARNINGS_AS_ERRORS=OFF -DNS3_NATIVE_OPTIMIZATIONS=OFF -G Unix Makefiles .. ; cd ..
```

Figure 6.2: NS-3 build configuration.

not all of the modules will be required for our experiments, we will not build all of them. The configuration file `~/ns3-student/.ns3rc` is used to configure the list of modules that we want to build. In our case, we build the following modules:

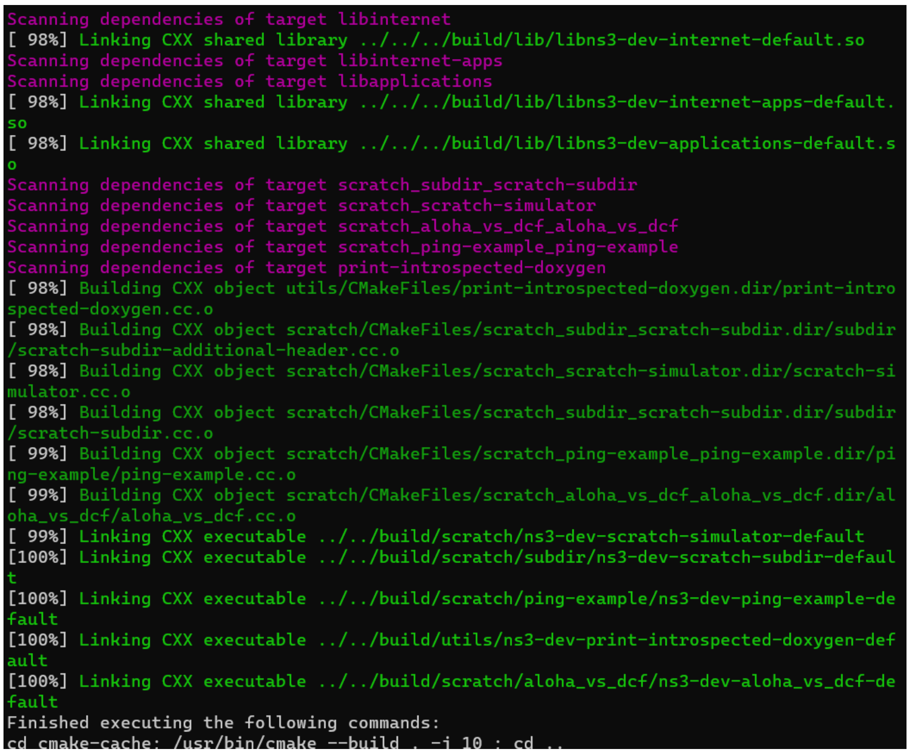
- applications (for simple UDP application that we will use to generate data);
- bridge (contains implementation of switch);
- config-store (to configure parameters of different modules);
- core (the core of the simulator itself);
- csma (for ethernet);
- internet (Internet Protocol (IP), User Datagram Protocol (UDP), Address Resolution Protocol (ARP) and other protocols of the internet stack);
- internet-apps (for ping);
- mobility (for setting positions of the devices);
- network (all main simulator abstractions are implemented in this module, i.e., packet, network device, header, etc.);
- propagation (for modeling of the physical channel);
- wifi (for implementation of wifi).

Build the project with the following command:

```
% ./ns3 build
```

Note that NS-3 will build using multiple threads, thus all or most of your CPUs will become busy. If you want to limit the number of parallel threads, e.g., to 10, use parameter -j:

```
% ./ns3 build -j 10
```



```
Scanning dependencies of target libinternet
[ 98%] Linking CXX shared library ../../../build/lib/libns3-dev-internet-default.so
Scanning dependencies of target libinternet-apps
Scanning dependencies of target libapplications
[ 98%] Linking CXX shared library ../../../build/lib/libns3-dev-internet-apps-default.
so
[ 98%] Linking CXX shared library ../../../build/lib/libns3-dev-applications-default.s
o
Scanning dependencies of target scratch_subdir_scratch-subdir
Scanning dependencies of target scratch_scratch-simulator
Scanning dependencies of target scratch_aloha_vs_dcf_aloha_vs_dcf
Scanning dependencies of target scratch_ping-example_ping-example
Scanning dependencies of target print-introspected-doxygen
[ 98%] Building CXX object utils/CMakeFiles/print-introspected-doxygen.dir/print-intro
spected-doxygen.cc.o
[ 98%] Building CXX object scratch/CMakeFiles/scratch_subdir_scratch-subdir.dir/subdir
/scratch-subdir-additional-header.cc.o
[ 98%] Building CXX object scratch/CMakeFiles/scratch_scratch-simulator.dir/scratch-si
mulator.cc.o
[ 98%] Building CXX object scratch/CMakeFiles/scratch_subdir_scratch-subdir.dir/subdir
/scratch-subdir.cc.o
[ 99%] Building CXX object scratch/CMakeFiles/scratch_ping-example_ping-example.dir/pi
ng-example/ping-example.cc.o
[ 99%] Building CXX object scratch/CMakeFiles/scratch_aloha_vs_dcf_aloha_vs_dcf.dir/al
oha_vs_dcf/aloha_vs_dcf.cc.o
[ 99%] Linking CXX executable ../../build/scratch/ns3-dev-scratch-simulator-default
[100%] Linking CXX executable ../../build/scratch/subdir/ns3-dev-scratch-subdir-defaul
t
[100%] Linking CXX executable ../../build/scratch/ping-example/ns3-dev-ping-example-de
fault
[100%] Linking CXX executable ../../build/utils/ns3-dev-print-introspected-doxygen-def
ault
[100%] Linking CXX executable ../../build/scratch/aloha_vs_dcf/ns3-dev-aloha_vs_dcf-de
fault
Finished executing the following commands:
cd cmake-cache; /usr/bin/cmake --build . -j 10 ; cd ..
```

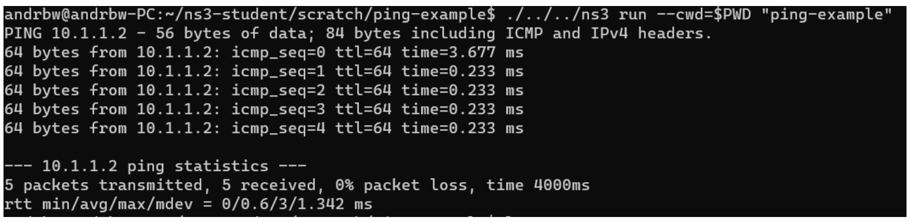
Figure 6.3: NS-3 build execution.

When the command finishes execution, you should see output on command line similar to the one on Fig. 6.3.

3. To test if the build process was successful, we will run a test experiment. Go to directory `~/ns3-student/scratch/ping-example` and execute the following command (the PWD command returns the full pathname of the current working directory):

   ```
   % ./../../ns3 run --cwd=$PWD "ping-example"
   ```

   Here we use the parameter cwd to specify the working directory, which NS-3 will use for all relative paths, e.g., to read from files or print to them. By default, the working directory is the root directory of the NS-3 project. If the experiment is finished correctly, you will see the same output as in Fig. 6.4. Besides, multiple pcap traces will appear in the directory.



```
andrbw@andrbw-PC:~/ns3-student/scratch/ping-example$ ./../../ns3 run --cwd=$PWD "ping-example"
PING 10.1.1.2 - 56 bytes of data; 84 bytes including ICMP and IPv4 headers.
64 bytes from 10.1.1.2: icmp_seq=0 ttl=64 time=3.677 ms
64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=0.233 ms
64 bytes from 10.1.1.2: icmp_seq=2 ttl=64 time=0.233 ms
64 bytes from 10.1.1.2: icmp_seq=3 ttl=64 time=0.233 ms
64 bytes from 10.1.1.2: icmp_seq=4 ttl=64 time=0.233 ms

--- 10.1.1.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4000ms
rtt min/avg/max/mdev = 0/0.6/3/1.342 ms
```

Figure 6.4: Output generated by ping-example experiment run.

Exercise 2: Building a simple **wired** topology and executing the ping command

In this exercise, we will go deeper into the ping-example experiment. Specifically, we will learn how to create a simple wired topology in NS-3 and execute a ping command to initiate an exchange of Internet Control Message Protocol (ICMP) packets between the nodes. The network setup in Fig. 6.5 and Tab. 6.1 is used in this exercise.

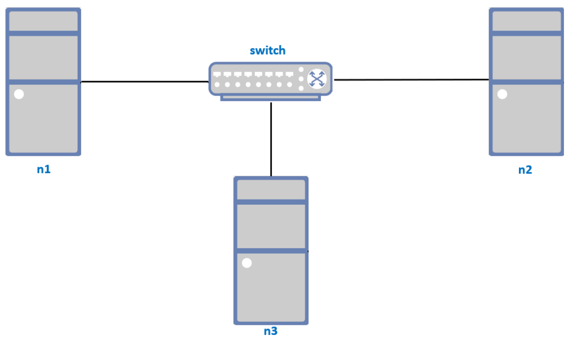


Figure 6.5: Wired network topology in ping-example experiment.

End node	IP address
n1	`10.1.1.1/24`
n2	`10.1.1.2/24`
n3	`10.1.1.3/24`

Table 6.1: IP addresses of the hosts.

1. When we run an experiment in NS-3, we can set its parameters via the command line. For that, the class CommandLine is used. In experiment ping-example.cc, we have the following structure of the command line arguments:

```
CommandLine cmd;
cmd.AddValue("interval", "The time to wait between two packets",
    interPacketInterval);
cmd.AddValue("size", "Data bytes to be sent, per-packet", size);
cmd.AddValue("count", "Number of packets to be sent", count);
cmd.AddValue("srcIdx",
"End node index that is ping source, e.g., if 1 the src IP is \"10.1.1.1\"",
srcIdx);
cmd.AddValue("dstIdx",
"End node index that is ping source, e.g., if 1 the src IP is \"10.1.1.2\"",
dstIdx);
cmd.Parse(argc, argv);
```

Each function call AddValue has three arguments: the parameter name, its text description and reference to a variable to be set. For any experiment, we can always see the description of its variables by executing the run command with argument PrintHelp:

```
% ./../../ns3 run --cwd=$PWD "ping-example --PrintHelp"
```

If we do not specify any arguments after ping-example, the variables will not be changed. That means that their default values will be used. If we want to change the value of a particular variable, we can always specify its value in the run command:

```
% ./../../ns3 run --cwd=$PWD "ping-example --count=10"
```

L6-2-1 Using the description of the command line arguments and the output of the experiment run, describe what the experiment does by default. /1

2. To create the devices and connect them with links, we use multiple NS-3 abstractions. The basic computing device abstraction is called Node. When you want to connect a real non-simulated computer to a network, you have to use a Network Interface Card (NIC). To control this hardware, network device software drivers are used. In NS-3, the NetDevice abstraction is used to cover both the software driver and the simulated NIC hardware. A NetDevice is “installed” in a Node in order to enable the Node to communicate with other Nodes in the simulation via Channels. Just as in a real computer, a Node may be connected to more than one Channel via multiple NetDevices. Just as an Ethernet NIC is designed to work with an Ethernet network, the CsmaNetDevice is designed to work with a CsmaChannel and a WifiNetDevice is designed to work with a WifiChannel.

   In our experiment, we will use CsmaHelper to create links between pairs of devices. Each link that we create has 100 Mbps data rate and 50µs delay. Since we have three links, in total we will create 6 NetDevices: 1 installed on each of the end nodes and 3 installed on the switch device. To enable switch capabilities on a switch node, we also use BridgeHelper. The following code is responsible for the described procedure:

```
//create nodes {n1, n2, n3}
NodeContainer endNodes;
endNodes.Create(3);

//create switch node
NodeContainer switchNode;
switchNode.Create(1);

//set parameters for Ethernet link
CsmaHelper ethernet;
ethernet.SetChannelAttribute("DataRate", StringValue("100Mbps"));
ethernet.SetChannelAttribute("Delay", StringValue("50us"));

//insert Ethernet NICs into nodes and switch
NetDeviceContainer endNodeDevices;
NetDeviceContainer switchDevices;
for (int i = 0; i < 3; i++)
{
NetDeviceContainer linkDevices = ethernet.Install(NodeContainer (endNodes.Get (
    i), switchNode));
endNodeDevices.Add (linkDevices.Get (0));
switchDevices.Add (linkDevices.Get (1));
}

//Enable switch capabilities on switch interfaces
BridgeHelper bridge;
bridge.Install(switchNode.Get (0), switchDevices);
```

   L6-2-2 If we transmit a 100 bytes packet from node n1 to n2, and node n2 immediately transmits the same packet back to n1, what will be the round-trip time? /1

3. In this experiment, the exact positions of the nodes are not important. For simplicity, we place all of them to the same point: (0, 0, 0). For that, we use ListPositionAllocator with only one position in its list. Then, on each installation it will set this position to each node. Since we do not want the nodes to move, we use ConstantPositionMobilityModel. The following code is responsible for the described procedure:

```
    MobilityHelper mobility;
    Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>
        ();

    //place all nodes at (0, 0, 0)
    positionAlloc->Add (Vector (0.0, 0.0, 0.0));
    mobility.SetPositionAllocator (positionAlloc);

    //all nodes do not move, i.e., have constant position
    mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
    mobility.Install (endNodes);
    mobility.Install (switchNode);
```

4. To enable support of all main internet protocols, we use InternetStackHelper. Then, we use Ipv4AddressHelper to assign IP version 4 (IPv4) addresses to all nodes.

```
    //install internet protocols
    InternetStackHelper internet;
    internet.Install(endNodes);

    //Assign IP addresses from subnet 10.1.1.0/24
    Ipv4AddressHelper ipv4;
    ipv4.SetBase("10.1.1.0", "255.255.255.0");
    ipv4.Assign(endNodeDevices);
    Ipv4InterfaceContainer interfaces = ipv4.Assign(endNodeDevices);
```

5. Finally, we install the ping tool on one of the nodes (with index srcIdx) where we want to execute the ping command. We set the following parameters: destination IP address, the size of the ICMP packets, the interval between packets, the number of packets to be sent (using the parameter “Count”), the start time and the end time of the ping command execution. We also enable pcap files collection so that we can analyse the traffic captured on all devices during the experiment.

```
    //Configure ping from node srcIdx to node dstIdx
    PingHelper pingHelper(interfaces.GetAddress (dstIdx), interfaces.GetAddress (
        srcIdx));
    pingHelper.SetAttribute("Interval", TimeValue(interPacketInterval));
    pingHelper.SetAttribute("Size", UintegerValue(size));
    pingHelper.SetAttribute("Count", UintegerValue(count));

    //Install ping tool on source node
    ApplicationContainer apps = pingHelper.Install(endNodes.Get(srcIdx));
    apps.Start(Seconds(1));
    apps.Stop(Seconds(50));

    //Capture traffic from all interfaces
    ethernet.EnablePcapAll("ping-example");
```

❗ Note that we do not install the ping tool on the receiver side. The tool is responsible for sending the ICMP echo requests, and output generation after interpreting the echo replies. However, generation of echo reply on a received echo request is a responsibility of ICMP protocol itself, so this procedure does not require the ping tool installation.

L6-2-3 How many pcap files will be generated by the experiment? Why? /1

6. The last part is often the same in all experiments. It consists of three commands: Stop, Run and Destroy. The Stop command creates the terminating event at a specific time moment that corresponds to the end of simulation. The Run command starts simulation, i.e., starts processing of the first event in the events queue. Finally, the Destroy command destroys all the objects that have been created during the experiment.

```
    Simulator::Stop(Seconds(60.0));
    Simulator::Run();
    Simulator::Destroy();
```

7. Now, when you understand the structure of the experiment, run the experiment with the following parameter configuration: send 8 ICMP ping packets from node with IP address 10.1.1.3 to node with IP address 10.1.1.1, set interval between packets to 0.5 seconds and packet size to 56 bytes. **Do not modify any source file!** Save the program output and the generated pcap files.

   L6-2-4 Which command did you use to run the experiment? /1

   L6-2-5 Look inside the content of each pcap file. Which packets do you see inside them? Do all the pcap files contain the same packets? Why? /1

8. Modify the source file of the experiment, `ping-example.cc`: add function call ArpCache::PopulateArpCache () right after the IPv4 addresses assignment.

```
    Ipv4AddressHelper ipv4;
    ipv4.SetBase("10.1.1.0", "255.255.255.0");
    ipv4.Assign(endNodeDevices);
    Ipv4InterfaceContainer interfaces = ipv4.Assign(endNodeDevices);

    ArpCache::PopulateArpCache ();

    //Create Ping application and installing on node
```

   Run the experiment from the previous task again with the same parameters. Note that before running the experiment NS-3 will rebuild the file, because we changed it. Save the program output and the generated pcap files.

   L6-2-6 Is there any difference in program output and pcap files compared to the previous task? What does the function ArpCache::PopulateArpCache () do to cause such difference? /2

Exercise 3: Building simple **wireless** topology and executing the ping command

In this exercise, we will modify the experiment file `ping-example.cc`. Specifically, we will substitute the wired topology with the wireless one. We will consider an older Wi-Fi wireless technology, that is standardized as IEEE 802.11a. The newest Wi-Fi standard is IEEE 802.11ax, that is commonly known as Wi-Fi 6. However, for simplicity we will configure an older version, it will be enough for our investigation purposes.

Since we will make all devices capable of communicating via wireless medium, we will no longer need the switch. Wi-Fi networks can operate in two modes: infrastructure and ad-hoc. Infrastructure mode is a very common mode that we are using at our homes when we connect all Wi-Fi devices to an access point. However, the presence of the access point is not necessary, because the devices can communicate with each other directly. For the sake of simplicity, we will consider this second mode, called **ad-hoc**. Then the network topology becomes very simple (see Fig. 6.6).

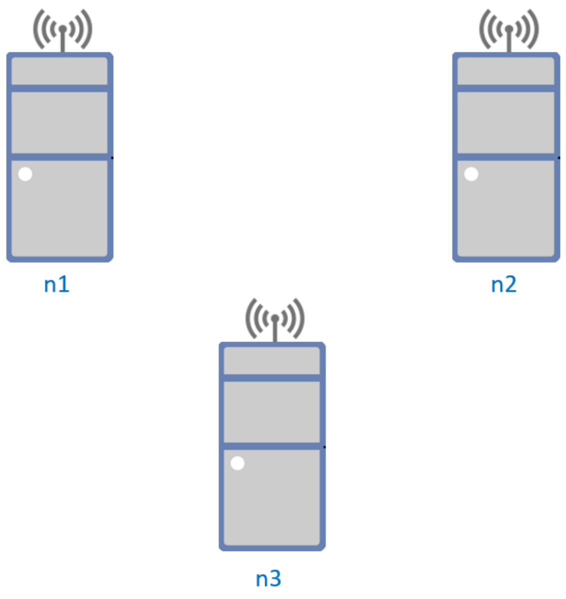


Figure 6.6: Wireless network topology in ping-example experiment.

1. Delete from the code everything related to the switch and wired links configuration. Note that pcap traces collection also has to be replaced, because we will not use the CsmaHelper anymore.

2. Now we will add the Wi-Fi network configuration and install Wi-Fi NetDevices on each node. For that, we will use the following code:

```
//configure Wi-Fi parameters
WifiHelper wifi;
wifi.SetStandard (WIFI_STANDARD_80211a);
SpectrumWifiPhyHelper wifiPhy;
Ptr<MultiModelSpectrumChannel> channel = CreateObject<MultiModelSpectrumChannel
    >();
wifiPhy.SetChannel (channel);
WifiMacHelper wifiMac;
wifiMac.SetType ("ns3::AdhocWifiMac", "QosSupported", BooleanValue (false));
wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
                              "DataMode",StringValue ("OfdmRate6Mbps"));

//insert Wi-Fi NICs into nodes
NetDeviceContainer endNodeDevices = wifi.Install (wifiPhy, wifiMac, endNodes);
```

Here we configure separately Physical (PHY) and Media Access Control (MAC) layers of Wi-fi and pass them as arguments to the Install function of WiFiHelper. We explicitly set the constant rate mode, because by default Wi-Fi adapts its transmission rate depeding on nodes' channel conditions. We disable this feature and fix the transmission rate to 6 Mbps, which coresponds to the lowest Wi-Fi data rate. To make this code work, we have to include two libraries: `ns3/wifi-module.h` and `ns3/multi-model-spectrum-channel.h`.

To enable pcap traces collection, instead of the prior line of code we add:

```
wifiPhy.EnablePcapAll("ping-example")
```

3. Run the following command:

   ```
   % git diff
   ```

   You will see highlighted the differences you made in the file. Save the output of the command to a text file.

4. Run the modified experiment. Make sure that the ArpCache::PopulateArpCache function call is added into the experiment source file. Save the program output and the generated pcap files.

   L6-3-1 Look inside the content of each pcap file. Which packets do you see inside them? Do all the pcap files contain the same packets? Why? /2

   L6-3-2 Look inside the content of the pcap file that corresponds to a source node that executes the ping command. How many packets are sent for each ICMP ping request? /2

   L6-3-3 Compare the results for the wireless and wired topologies. Use the pcap traces and saved text files. Please also indicate the name of the file that contains the output of the diff command. /3

Part 2. Modeling of pure Aloha

Starting from this section, we will no longer work with the ping-example experiment and switch to the aloha_vs_dcf experiment located in directory `~/ns3-student/scratch/aloha_vs_dcf/`. We will start with the modeling of the pure Aloha MAC layer protocol. Although Wi-Fi devices do not use Aloha for communication, NS-3 allows us to modify any protocol behavior, or substitute one protocol with another one.

Aloha is a random access protocol introduced in 1970s by Norman Abramson and his colleagues at the university of Hawaii (history of Aloha is well described by Abramson in http://www2.hawaii.edu/~pager/Aloha%20History.pdf). The idea is very simple: when a device has a packet for transmission, it can immediately start to transmit. However, when more than one device transmits at the same time, there is a high probability that none of the packets will be received successfully. Such an event is called a collision. When a device transmits a packet, it expects an acknowledgment is sent back from the receiver. If the acknowledgment does not arrive at the sender within a timeout, it assumes that the packet was not successfully delivered, e.g., due to collision. In this case, the sender retransmits the packet after some randomly selected amount of time. This delay should be random to avoid further collisions between retransmissions.

In telecommunication theory, the time intervals between packet arrivals are often modeled with an exponential random variable. Since the retransmissions are sent after a random delay, we can further assume that the aggregated data flow of both initial transmissions and retransmissions for each user is a single flow with inter-packet intervals distributed exponentially. In general, this is not true, but this assumption can provide a good trade-off between modeling accuracy and complexity. We say that the data flow has intensity λ packets per second (pkts/s) if the average inter-packet interval is $\frac{1}{\lambda}$. There is a very useful property: if we have n flows with intensities λ_1, λ_2, ..., λ_n, then the overall intensity equals $\sum_{i=1}^{n} \lambda_i$. Such data flows are called Poisson data flows.

In this section and further, we consider the following scenario: N “client” devices that generate Poisson data flows of the same intensity λ destined to the same single “server” device. Since we aggregate initial transmission and retransmissions into a single Poisson flow, we assume that the packets are dropped if they have not been successfully delivered, i.e., there is only one transmission attempt for each packet. Furthermore, we assume that transmissions are failed only due to collisions, i.e., when two or more devices transmit data simultaneously. There is a special type of collisions that we should keep in mind, “internal” collisions, when a packet is generated at a device before this devices finishes transmission of the previous packet. In this case, we also drop the second packet. The effect of “internal” collisions is negligible if the traffic intensity on each node is low. In each experiment, for a given number of devices we calculate the aggregate network throughput. Then, we can plot the dependency of the aggregate throughput on the load (intensity of aggregate flow from all nodes).

Exercise 4: Simulation of pure Aloha using Wi-Fi module

In this exercice, we will model the pure Aloha protocol efficiency in the described above scenario.

1. Run experiment aloha_vs_dcf with its default parameters. By default, the experiment creates 2 nodes, client and server.

```
% ./../../ns3 run --cwd=$PWD "aloha_vs_dcf"
```

L6-4-1 How many additional files appeared in the working directory? What kind of files? What is the content of the text file? /1

L6-4-2 Look inside the content of each pcap file. Which transport layer protocol is used by an application that generates packets? What are the packet size and payload size? /2

L6-4-3 How much time does it take to transmit one packet? Use wireshark to answer. /2

Note that wireshark shows you the time elapsed since the first captured packet. To see the difference between timestamps in two files, go to View -> Time Display Format -> Time of Day.

2. Run experiment aloha_vs_dcf two more times, but this time with additional parameter RngRun.

```
% ./../../ns3 run --cwd=$PWD "aloha_vs_dcf --RngRun=1"
% <...>
% ./../../ns3 run --cwd=$PWD "aloha_vs_dcf --RngRun=2"
```

Compare the content of the text file for the first and second run. You will notice that they are different. Moreover, if you will open the file `aloha_vs_dcf.cc`, you will see that there is no such command line argument defined. This is because this argument is global and does not need explicit definition in a scenario. Changing of this argument changes the generation sequence for all random variables used during the experiment (see more details at https://www.nsnam.org/docs/manual/html/random-variables.html). For example, the exponential random variables responsible for inter-packet intervals will generate different values for different values of RngRun argument. Usually, in experiments that involve randomness, the experiment is executed multiple time with different runs (RngRuns), and the results of different runs are averaged.

3. Now look deeper inside the file `aloha_vs_dcf.cc` and find the differences between this file and `ping-example.cc` with configured wireless topology. You will notice multiple things that you haven't seen before. We will go through all of those differences. First, there are additional configurations right after the definition of command line arguments.

```
    // disable fragmentation for frames below 2200 bytes
    Config::SetDefault ("ns3::WifiRemoteStationManager::FragmentationThreshold",
        StringValue ("2200"));
    // turn off fragmentation at IP layer
    Config::SetDefault ("ns3::WifiNetDevice::Mtu", StringValue ("2200"));
    // turn off RTS/CTS for frames below 2200 bytes
    Config::SetDefault ("ns3::WifiRemoteStationManager::RtsCtsThreshold",
        StringValue ("2200"));

    //Allow only one transmission attempt
    Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSsrc", UintegerValue (0)
        );
    Config::SetDefault ("ns3::WifiRemoteStationManager::MaxSlrc", UintegerValue (0)
        );

    //Disable backoff for Aloha
    if (!isDcf)
      {
        Config::SetDefault ("ns3::Txop::DisableBackoff", BooleanValue (true));
      }
```

With this configurations, we avoid fragmentation of packets and disable an exchange of additional specific Wi-Fi packets. We also allow only one transmission attempt for each packet, so that we drop packets if the intial transmission fails. The flag isDcf switches the MAC protocol between Aloha and DCF. When this variable is false, we also disable the backoff procedure, which is relevant only for DCF.

4. Second, we install applications on the nodes to transmit and receive data. For each client node, we install the UdpEchoClient application, whose source code is located in `~/ns3-student/src/applications/model/udp-echo-client.cc`. We configure this application to send an unlimited number of packets (UINT32_MAX is a sufficiently big number), i.e., until the end of simulation. We also configure the payload size and inter-packet interval. On the server node, we install the PacketSink application, whose source code is located in `~/ns3-student/src/applications/model/packet-sink.cc`. This application simply receives all the packets that are destined to it, i.e., to the right IP address and port number.

```
    //Install server on station 0
```

```
PacketSinkHelper server ("ns3::UdpSocketFactory",InetSocketAddress(interfaces.
    GetAddress (0),9));

ApplicationContainer serverApps = server.Install (serverStation);
serverApps.Start (serverStart);
serverApps.Stop (simTime);

//Install clients
UdpEchoClientHelper echoClient (interfaces.GetAddress (0), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (UINT32_MAX));
echoClient.SetAttribute ("EnableRandomInterval", BooleanValue (true));
std::ostringstream intervalDist;
intervalDist << "ns3::ExponentialRandomVariable[Mean=" << packetInterval.
    GetSeconds () << "]";
echoClient.SetAttribute ("RandomIntervalVariable", StringValue (intervalDist.
    str ()));
echoClient.SetAttribute ("PacketSize", UintegerValue (packetSize));

ApplicationContainer clientApps = echoClient.Install (clientStations);
clientApps.Start (clientStart);
clientApps.Stop (simTime);
```

L6-4-4 On which port number does the PacketSink application listen to incoming data? How can you find this out? /1

5. Third, we use the NS-3 tracing system to catch the events when packets arrive to the PacketSink application (you can find more information about tracing at `https://www.nsnam.org/docs/tutorial/html/tracing.html`).

```
uint32_t RcvPktCount = 0;

void
ReceiveTrace (Ptr<const Packet> pkt, const Address & addr)
{
RcvPktCount++;
}

<...>

serverApps.Get(0)->TraceConnectWithoutContext ("Rx", MakeCallback(&ReceiveTrace
    ));
```

When we connect a trace to an NS-3 object (here to PacketSink application), we call a specific function (here ReceiveTrace) every time a certain event happens at this object. To see how it works, open the file `~/ns3-student/src/applications/model/packet-sink.cc` and find in function PacketSink::GetTypeId() the definition of the attribute "Rx". The trace source connected to this attribute is called m_rxTrace. Now locate the place in this file, when this trace source is called.

```
void
PacketSink::HandleRead(Ptr<Socket> socket)
{
  <...>
  while ((packet = socket->RecvFrom(from)))
  {
    m_rxTrace(packet, from);
  }
```

```
        <...>
    }
```

Every time when the application reads something from the socket, it calls m_rxTrace, which is further connected with our function ReceiveTrace. Note that there are two parameters passed to m_rxTrace: the packet received from the socket and the address of its source. Exactly the same parameters are expected by the function ReceiveTrace (we do not use them, but we could).

6. Finally, we write a function PrintProgress that schedules itself once per second (in simulation time, not real time) and prints in standard error stream the percentage of time passed from the start of the experiment. It is useful especially for long experiments when we want to understand how fast the experiment runs and how much time is left till the end of the experiment. Take a look at the syntax of the Simulator::Schedule function:

```
Simulator::Schedule (Seconds (1), &PrintProgress, simTime);
```

In this case, it takes three arguments. The first argument is the delay between the current simulation time (Simulator::Now()) and the target time when you plan to execute the event. The second argument is the reference to the function to be exectuted. Note that function PrintProgress in not a member of any object. If we have to schedule a function call for a member of an object, we should add an additional argument after the reference to this member – the pointer to the object, owner of this member. All the following arguments will be passed to the scheduled function. For example, function PrintProgress expects one argument, the simulation time. Thus we pass this argument to the Simulator::Schedule. See more details at `https://www.nsnam.org/docs/manual/html/events.html`.

7. Now, when you understand the code and did some test runs, it is time to conduct our first meaningful experiment. Using the experiment aloha_vs_dcf and varying the number of stations from 1 to 100 with step size 10 (i.e., 1, 11, 21, 31, etc.), plot throughput as a function of the number of devices. Average results over 5 runs with RngRun=1, 2, 3, 4, 5. Save all the scripts you used to run experiments and plot the results.

L6-4-5 Describe the dependency of throughput on the number of devices. Include the figure. /2

By default, the pcap traces collection is enabled. We will not need them, so we can disable it by setting the flag collectPcap to false. Besides, to avoid rebuilding of the source file you can use the flag "–no-build".

Multiple experiments with different parameters can be easily run automatically using a bash script, e.g., the following one (saved in `~/ns3-student/scratch/aloha_vs_dcf/scripts/run_experiment.sh`):

```
#!/bin/bash
set -e

for run in $(seq 1 5); do
  for n in $(seq 1 10 100); do
    ./../../ns3 run --no-build --cwd=$PWD "aloha_vs_dcf --RngRun=$run --
        numOfStations=$n --isDcf=false --collectPcap=false"
    cat result.txt >> aloha-pure-$run.dat
    done
done
```

Tip: do not run all the experiments when you are not sure that everything works correctly. Check on RngRun=1 and several values of n. You can use one laptop for running the experiments, and another one for programming.

Note that we have to run the experiment 5×20 times, so it might take too long if we use only one thread. To benefit from multiple threads, you can use GNU Parallel. For example, you can use a script similar to the following one (saved in `~/ns3-student/scratch/aloha_vs_dcf/scripts/run_experiment-parallel.sh`):

```
#!/bin/bash
set -e

mkdir -p results
parallel "./../../ns3 run --no-build --cwd=$PWD \"aloha_vs_dcf --RngRun={1} --
    numOfStations={2} --isDcf=false --alohaSlot=0us --collectPcap=false --
    outFileName=results/{1}-{2}.txt\"" ::: $(seq 1 5) ::: $(seq 1 10 100)
```

To plot the results, you can use the matplotlib Python library, or any other instrument. For example, the following script can be used (saved in `~/ns3-student/scratch/aloha_vs_dcf/scripts/plot.py`):

```
#!/usr/bin/python3

import numpy as np
import matplotlib.pyplot as plt

num_runs = 5
max_stas = 100

pure = np.zeros (10)
for run in range (1, num_runs+1):
df_pure = np.genfromtxt (f"aloha-pure-{run}.dat", names=None)
pure += df_pure[:, 1] / num_runs

plt.figure (figsize=[5.5, 4.0])

num_stas = np.array (range(1, max_stas + 1, 10))
plt.plot (num_stas, pure, label='sim pure Aloha', color = 'g', marker = '+',
    linestyle='None')

plt.xlabel ("Number of stations")
plt.ylabel ("Throughput, Mbps")
plt.grid ()
plt.legend (loc="best")

plt.savefig ("throughput.png", dpi=200)
```

If you use GNU parallel to run experiments, then before executing the script plot.py you can run a simple bash script (saved in `~/ns3-student/scratch/aloha_vs_dcf/scripts/merge_results.sh`) to gather the results:

```
#!/bin/bash

set -e
```

```
    for run in $(seq 1 5); do
      touch aloha-pure-$run.dat
      for n in $(seq 1 10 100); do
        cat results/pure-$run-$n.txt >> aloha-pure-$run.dat
      done
    done
```

General remark: **do not** delete any results you calculate during the lab. You will need the same results in several parts, and some of them require much computation time.

8. From theory, it is well known that the dependency of normalized throughput on normalized load can be expressed with the following equation: $T = G \cdot \exp^{-2G}$ [1]. In this equation, normalized load G is the average number of packets that are generated during the transmission of one packet, while normalized throughput T is a fraction of time when packets are successfully transmitted without collisions. Rescale the figure plotted in the previous task according to the described axis and plot **on the same figure** two curves: (1) obtained from simulation, and (2) according to the analytical equation. Note that you do not need to recalculate the results, just rescale the ones obtained in the previous task. Save the modified scripts.

L6-4-6 Describe how you rescale the results. Refer to the modified scripts. /2

L6-4-7 Compare the curves obtained from the analytical equation and from simulation. What is the maximum value of the observed normalized throughput? At which point is it achieved? Include the figure. /2

L6-4-8 In our experiment, we fix the traffic intensity on each node and vary the load by changing the number of devices. We could also vary the load in a different way: fix the number of devices, e.g., to 10, and vary the intensity of traffic. Will this approach provide the same results? Explain why (not). /2

Here are some tips regarding rescaling of the figure. Let the number of stations be N, the average interval between packets be $\hat{t}$ µs, and the packet transmission duration (from the Task L6-4-3) be τ µs. Then, the normalized load equals $G = \frac{N \cdot \tau}{\hat{t}}$. Further, if the network throughput equals S Mbps, and the packet size equals p bits, then the normalized throughput equals $T = \frac{S \cdot \tau}{p}$.

Do not forget to include all the scripts you use in the report. Refer to these scripts in your answers so that it will be easy to find them.

Do not forget to rebuild the project if you change something in the source files, e.g., in `aloha_vs_dcf.cc`.

Part 3. Modeling of slotted Aloha

In pure Aloha, devices transmit data completely asynchronously. It is intuitively clear that synchronization between devices can improve the network capacity. For example, if we divide time into slots, where the slot duration equals the duration of a packet transmission, and allow devices to transmit only in the beginning of these slots, the collisions will occur only when multiple devices select the same slot. In other words, the transmission from a device cannot interrupt the ongoing transmission in the middle of the slot. Lawrence G. Roberts was the first researcher who derived the analytical equation to calculate such a system's throughput: $T = G \cdot \exp^{-G}$ [2]. We can see that the exponent becomes -1 instead of -2, thus the throughput for slotted Aloha is indeed higher.

Exercise 5: Simulation of slotted Aloha using the Wi-Fi module

In this exercice, we will model the slotted Aloha protocol efficiency. For that, we will modify the implementation of the Aloha protocol in NS-3, i.e., we will modify the source file in the Wi-Fi module.

1. Before this exercise, we worked mostly in the scratch directory, where the source files of the experiments are located. Now it is time to look inside the implementation of the Aloha protocol. Open the file `ns3-student/src/wifi/model/adhoc-aloha-mac.cc`. It has multiple functions, but we are interested only in two of them, Enqueue and Receive. The first function is responsible for data transmission from the device to the wireless medium, while the second one – for data reception from the wireless medium.

   Not all the content of these functions is important for us. Below you can find the important parts of the Enqueue function:

```
void
AdhocAlohaMac::Enqueue(Ptr<Packet> packet, Mac48Address to)
{
  if (!GetWifiPhy ()->IsStateTx ()) //if already transmitting - drop packet
  {
  WifiMacHeader hdr;
  hdr.SetAddr1(Mac48Address::GetBroadcast ());
  <filling the rest of the header>

  Ptr<WifiMpdu> mpdu = Create<WifiMpdu>(packet, hdr);
  <...>
  GetTxop ()->Enqueue (mpdu);
  <...>
  //immediately start transmission
  GetFrameExchangeManager ()->StartTransmission (GetTxop (), width);
  }
}
```

   The code can be interpreted as follows. Whenever a device has a packet for transmission:

   - If it is already transmitting something, then we experience an internal collision. According to our previously negotiated assumptions, we expect the device to drop the packet.
   - If it does not transmit anything yet, then we can immediately start transmission of the packet.

   You probably already noticed in the pcap files that there is something different compared to the pcap file from the ping-example experiment – there are no Wi-Fi acknowledgements. Normally, Wi-Fi devices acknowledge packets that they successfully receive. However, they do not do that for broadcast transmissions, otherwise all devices will have to send acknowledgements at the same time. We disable acknowledgements to allow traffic to go only in one direction, from clients to packet sink. That is why we set the destination address to Mac48Address::GetBroadcast (), which is ff:ff:ff:ff:ff:ff.

Now look at the Receive function. Similarly, below you can see the important parts:

```
void
AdhocAlohaMac::Receive(Ptr<const WifiMpdu> mpdu, uint8_t linkId)
{
  const WifiMacHeader* hdr = &mpdu->GetHeader();
  Mac48Address from = hdr->GetAddr2();
  Mac48Address to = hdr->GetAddr1();

  //filter broadcast packets
  if (hdr->IsData() && GetAddress () == Mac48Address ("00:00:00:00:00:01"))
  {
    ForwardUp(mpdu->GetPacket()->Copy(), from, to);
    return;
  }
}
```

Since we indicate the broadcast MAC address as a destination address, all the nodes will assume that each packet is destined to them. We know that we installed a packet sink on the first node, which has MAC address 00:00:00:00:00:01. Thus, we allow only this node to forward packets further up to the IP and UDP layers. This filtering as well as substitution of MAC address is not a general behavior and is done here only to disable Wi-Fi acknowledgements.

2. To implement slotted Aloha, we have to modify the Enqueue function. Specifically, we cannot send packets right away anymore. Instead, we have to defer transmissions (i.e., using the function call GetFrameExchangeManager ()->StartTransmission) until the closest slot boundary. To do that, we can use the function Simulator::Schedule, which we have already seen. Modify the Enqueue function to implement the described behavior. Here are several tips:

   - The call of Simulator::Schedule can look similar to this line:

   ```
   Simulator::Schedule (delay, &FrameExchangeManager::StartTransmission,
       GetFrameExchangeManager (), GetTxop (), width);
   ```

   - To get the current simulation time, you can use the function Simulator::Now (). Note that this function returns a Time object, which has function-members GetSeconds(), GetMilliSeconds(), GetMicroseconds(), etc. All these function-members, except GetSeconds(), return integer values.
   - The duration of the slot equals the duration of the packet transmission. You can use it as a constant.

   When you finish, go to the directory `ns3-student/src/wifi/model/` and execute the following command:

   ```
   % git diff adhoc-aloha-mac.cc
   ```

   Save the output of this command to a text file.

3. Rebuild the project. Using the experiment aloha_vs_dcf and varying the number of stations from 1 to 100 with step 10, plot the normalized throughput as a function of the normalized

load. Average results over 5 runs with RngRun=1, 2, 3, 4, 5. Save all the scripts you used to run experiments and plot the results. Plot **on the same figure** two curves: obtained from simulation and according to the analytical equation.

L6-5-1 Describe which modifications were required for the slotted Aloha implementation. Refer to your diff file. /1

L6-5-2 Compare curves obtained from the analytical equation and from simulation. What is the maximum value of the observed normalized throughput? At which point is it achieved? Include the figure. /1

L6-5-3 How would the throughput change if you set the slot size duration shorter than the packet transmission duration? If you set it longer? Explain your answers. /2

Part 4. Comparison between Aloha and DCF

In this final part of the lab, we will compare the performance of the Aloha protocol to DCF. A modified version of DCF is used in real Wi-Fi devices.

Exercise 6: Comparison between Aloha and DCF: can we do better than Aloha?

Switch from AdhocAlohaMac to AdhocWifiMac. Using the experiment aloha_vs_dcf and varying the number of stations from 1 to 100 with step 10, plot the estimated throughput as a function of the number of devices. Average results over 5 runs with RngRun=1, 2, 3, 4, 5. Save all the scripts you used to run the experiments and plot the results. Plot **on the same figure** three curves: pure Aloha, slotted Aloha and DCF (provided by AdhocWifiMac).

L6-6-1 Compare the curves. Which one provides the highest throughput? Why? /3

Acronyms

ARP	Address Resolution Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPv4	IP version 4
MAC	Media Access Control
UDP	User Datagram Protocol

Bibliography

[1] Norman Abramson. The ALOHA system: Another alternative for computer communications. In *Proceedings of the November 17-19, 1970, fall joint computer conference*, pages 281–285, 1970.

[2] Lawrence G Roberts. ALOHA packet system with and without slots and capture. *ACM SIGCOMM Computer Communication Review*, 5(2):28–42, 1975.