BTI 3021: Networking Project - Sprint 1

Christian Grothoff

1 Introduction

For this sprint you will implement, document and test an Ethernet switch in userland under GNU/Linux.

While the driver and skeleton you are given is written in C, you may use *any* language of your choice for the implementation (as long as you extend the `Makefile` with adequate build rules). However, if you choose a different language, be prepared to write additional boilerplate yourselves.

How an Ethernet switch works is expected to be understood from the networking class. If not, you can find plenty of documentation and specifications on the Internet.

1.1 Deliverables

There will be three main deliverables for the sprint:

Implementation You must **implement the switching algorithm**, extending the `switch.c` template provided (or write the entire logic from scratch in another language).

Testing You must implement and submit your own **test cases** by *pretending* to be the network driver (see below) and sending various Ethernet frames to your program and verifying that it outputs the correct frames. Additionally, you should perform and **document** *interoperability* tests against existing implementations (i.e. other notebooks from your team to ensure that your switch integrates correctly with other implementations).

Design and documentation You must design the main data structure (switching table), the switching algorithm and create a **comprehensive test plan**.

1.2 Functionality

We will specifically also look for the following properties of a switch:

- Support for multicast and broadcast
- Changing external connections (re-learning when devices move around the network)

- Managing an "attacker" process that sends from billions of MAC addresses. Ensure your switch's learning table uses finite memory.

1.3 Testing

For your test plans, please make sure to supply the following information for each test:

- Test Case ID — This field uniquely identifies a test case.
- Test case Description/Summary — This field describes the test case objective.
- Pre-requisites — This field specifies the conditions or steps that must be followed before the test steps executions.
- Test steps — In this field, the exact steps are mentioned for performing the test case.
- Expected Results — successful test
- Author — Name of the Tester.
- Automation — Whether this test case is automated or not.
- Status — Pass/fail with date and code version.

Note that in addition to automated tests similar to the `public-test-switch` you **should** probably do integration tests in a real network (evaluating manually with ping, wireshark, etc.). This will help you find problems in your implementation and ensure that your understanding of the network protocols is correct. Such integration tests should be documented and will be graded as part of the technical documentation.

1.4 Design and Documentation

The final documentation should include:

- Title page with group number and names/kurzel of all participants.
- Table of contents
- Introduction text (mention objectives and targeted audience)
- Data structure (switching table)
- Algorithm (switching algorithm) (use UML state diagram)
- Test strategy
- Product backlog and sprint backlog (Scrum)

- Glossary
- Bibliography

Text throughout must explain methods as well as justifications for choices made (why not other choices). The organisation should clearly indicate sections and sub-sections. The language should have a consistent voice with a fairly consistent register (semi-formal). The goal is clarity with minimal repetition. Concise language techniques (relative clauses, linking words...) are expected to be employed without forfeiting clarity. You should also use appropriate graphical representations and pay attention to how you visualize code. Algorithms and data structures should be presented in diagrams or with pseudo-code, but NOT in C syntax.

Documentation must be written in Markup and included in the Git, **but** must also be submitted by the deadline as a PDF to the respective Moodle assignment folder for grading by the English professors.

2 Grading

All deliverables must be submitted to Git (master branch) by the submission deadline announced on Moodle. The PDF files generated from the Markup documentation must **additionally be submitted via Moodle**.

You are expected to work in a team of **four students**. If needed, Prof. Grothoff may permit the creation of a team of 5 students or teams of 3 students. Each team is responsible for dividing up the work and coordinating as needed (SCRUM).

You can earn 16, 15 and 19 points in the three networking sprints, for a total of 50 points. You need **37 points** to pass the networking component of BTI 3021.

If your team is **for good reasons** smaller than four students, the passing threshold will be lowered by **2 points** per "missing" student per sprint. So a student going alone would still need **19 points** to pass.

Teams may **request** changes to team membership at the end of a sprint, but must provide a **justification** to the course coordinator, who may approve or decline the request.

2.1 Switch grading

For the **switch** sprint, you get points for each of the key deliverables:

Correct implementation	7
Comprehensive test cases	4
Technical documentation (A&D, test plan)	3
Quality of English in documentation	2
Total	16

2.1.1 Correct implementation

- 3 points for passing public test cases
- 4 points for passing secret test cases (see Section 6)

2.1.2 Comprehensive test cases

- 0 points if public reference implementation (see Section 6) fails test cases, **otherwise**
- 2 points for failing public buggy implementation (see Section 6), plus
- 2 points for finding bugs in secret buggy implementation(s)

If you believe you have found a bug in the provided reference implementation, you are encouraged to discuss it with the instructor. If you have found an actual bug in the reference implementation (that is within the scope of the assignment), you will be awarded a **bonus point** per acknowledged bug.

2.1.3 Technical documentation

- 1 point for good description of data structure
- 1 point for good description of switching algorithm
- 1 point for good description of test cases

Partial points may be awarded.

2.1.4 Quality of the English in the documentation

- 1 point for organisation / comprehension / consistency
- 1 point for correct grammar / spelling / punctuation

Partial points may be awarded.

3 Hardware

You will be given a 4-port Ethernet USB adapter that you can use to add four physical ports to any PC. If you use the laboratory PCs, be aware that some of the USB ports provide insufficient power. Which ones work is inconsistent even across identical PCs and often even the adjacent USB port works even though it looks identical!

You are not expected to write a driver to interact directly with the Ethernet USB adapter. Instead, you will use the provided `network-driver` which can already provide you with raw access to any Ethernet interface (incl. WLAN).

3.1 Alternative setup with virtual machines

Clone the Git repository at `https://gitlab.ti.bfh.ch/demos/vlab` and follow the provided instructions.

4 The `network-driver`

To access the hardware, your final program should be *executed* by the `network-driver`. For this, you call

```
$ network-driver IFC1 ... IFCn - ./switch ARGS
```

where "IFC1 ... IFCn" is the list of interface names that you want `network-driver` to support (i.e. "lan0", "lan1") and "PROG" is the name of your binary and "ARGS" are the command-line arguments to "switch". Note the "-" (single minus) between the last interface name and "switch". Also, "./switch" must be given with its path (i.e. "./switch" for the current working directory) or be located in a directory that is given in the "PATH" environment variable.

Once you start `switch` like this, you can read Ethernet frames and end-user commands from "stdin" and write Ethernet frames (and end-user output) to "stdout".

Note that you must follow the `network-driver`'s particular format for interprocess communication when reading and writing. You will **not** be communicating directly with the console!

4.1 Build the driver

To compile the code, run:

```
# This requires gcc
$ make
# Creating network interfaces requires 'root' rights
$ sudo chmod +s network-driver
# Try it out:
$ ./network-driver eth0 - ./parser
```

Press CTRL-C to stop the `network-driver` and `parser`.

4.2 Understanding the driver

The output of the driver is always in binary and generally in network byte order. You can use a tool like `hexer` to make the output slightly more readable.

The driver will always output a series of messages starting with a `struct GLAB_MessageHeader` that includes a type and a size.

When the driver starts, it first writes a control message (of type 0) with payload that includes 6 bytes for each of the local interface's MAC addresses to your `stdin`. Henceforce, messages received of type 0 will be single lines of

command-line input (including the '
n'-terminator, but excluding the 0-terminator of C) as typed in by the user.

Furthermore, the driver will output a `struct GLAB_MessageHeader` for each frame received. The `struct GLAB_MessageHeader` will be followed by the actual network frame, starting with the Ethernet frame excluding preamble, delimiter and FCS. The `struct GLAB_MessageHeader` includes the total length of the subsequent frame (encoded in network byte order, the size includes the `struct GLAB_MessageHeader`). The fixed message type identifies the number of the network interface, counting from one (also in network byte order).

In addition to writing received frames to your `stdin`, the driver also tries to read from your `stdout`. Applications must send the same message format to `stdout` that the driver sends them on `stdin`. The driver does **not** check that the source MAC is set correctly!

To write to the console's `stdout`, use a message type of 0. You may directly write to `stderr` for error messages.

5 Provided code

You are given a few C snippets as starting points. However, these mostly serve to *illustrate* how to process the output from the driver. You are completely free to implement your application in *any* programming language. Note that each file includes about 20 LOC of a licensing statement, so the functions provided should not provide a significant advantage for implementations in C.

sample-parser.c This file includes a simple starting point for a wireshark-like frame inspection code. It mostly shows how the frames are received and a bit how to use the other C files. (82 LOC)

glab.h A struct defining a MAC Address and a few common C includes. (90 LOC)

print.c This file shows how to wrap messages to print them via the driver. (112 LOC)

loop.c This could be the main loop of your application. The code includes some basic logic that looks at each frame, decides whether it is the MACs, control or an Ethernet frame and then calls the respective function. (93 LOC)

crc.c An implementation of checksum algorithms. (194 LOC)

If you are using another programming language, you are free to re-use an existing CRC implementation in that language.

The main file for the exercise is `switch.c`. In this file, you should implement a program `switch` which forwards frames received on any interface to any other interface, but passively learns MAC addresses and optimizes subsequent traffic.

6 Provided binaries

You are provided with several binaries:

public-test-switch A public test case, run using "./public-test-switch ./switch" to test your switch. Returns 0 on success.

public-switch Reference implementation of the "switch".

public-bug-switch Buggy implementation of a "switch".

7 Required make targets

You may modify the build system. However, the final build system must have the following `make` targets:

all build all binaries

clean remove all compiled files

switch build your "switch" binary from source; the binary MUST end up in the top-level directory of your build tree.

test-switch build your "test-switch" program from source; the program MUST end up in the top-level directory of your build tree.

check-switch Run "test-switch" against the "switch" binary.

For grading, we will basically run commands like:

```
GRADE=0
$ make test-switch
$ cp public-bug-switch switch
$ make check-switch || GRADE=‘expr $GRADE + 2‘
$ cp private-bug-switch switch
$ make check-switch || GRADE=‘expr $GRADE + 2‘
$ cp public-switch switch
$ make check-switch || GRADE=0
echo "Test grade: $GRADE"
```

You must thus make sure the build system continues to create programs in the right (top-level) location!

8 Warm-up exercises

These exercises may help you get familiar with the provided environment. They are **not graded**.

8.1 Frame parsing

Extend the simple frame `parser` to:

1. Output your system's MAC address(es) in the canonical human-readable format.

2. Output the source MAC, destination MAC, payload type and payload length of each frame received. Confirm your results with `wireshark`.

8.2 Implement a Hub

Implement `hub` which forwards frames received on any interface to all other interfaces (`eth0` through `eth3`), without changing them at all. The `hub` binary should take the list of interfaces to use on the command line.

9 Bonus exercise

This exercise may help you better understand the material from the networking course. It is not expected to be achievable within the period of the sprint and best done as a follow-up. It is **not graded**.

Implement *vswitch* which forwards frames received on any interface to any other interface, passively learns MAC addresses, and respects VLAN tags. As before, the command-line specifies the list of network interfaces you should switch on, but with additional options to specify the VLANS. Example:

```
$ network-driver eth0 eth1 eth2 eth3 - \
  vswitch eth0[T:1,2] eth1[U:1] eth2[U:2] eth3[U:2]
```

This is supposed to run VLANs 1 and 2 tagged on `eth0`, and VLANs 1, 2 or 2 untagged on `eth1`, `eth2`, or `eth3` respectively. Network interfaces specified without "[]" should operate untagged on VLAN 0. It is not allowed to have interfaces accept both tagged and untagged frames.

Test your implementation against the Netgear switch of the lab. Bridge a tagged VLAN (`VID`= 3) from the Netgear switch (`eth1`) with two untagged notebooks (`eth2`, `eth3`) using the BananaPi.

Properties to consider for a virtual switch:

- Correct forwarding? Do frames flow bidirectionally between eth2 and eth3?

- Correct switching? Is learning correctly implemented?

- Are VLAN tags added/stripped?

- Are correct limitations imposed on VLANs? For example using the configuration above, you might check that frames with $VID = 2$ are not forwarded to eth1.