pete > courses > CS 431 Spring 23 > Assignment 02

Assignment 02

If any of this is unclear, please get in touch with me ASAP. There are definitely things in the course I want you to struggle with. I absolutely do NOT want you struggling with understanding my expectations.

The goals of this assignment are you to cement your knowledge of how hosts, hubs, and switches behave with respect to Ethernet.

In Part I, you will write code that simulates the behavior of a host participating in an Ethernet network. You will build on this code in later assignments, to implement the upper layers of the protocol stack.

In Parts II and III, you will write programs that measure the behavior of hubs and switches under varying circumstances.

Due 12:00pm (noon) on Friday, 31 March.

Yes, this is almost three weeks. No, I do *not* count Spring Break as "time for you to work", so from my perspective it's really only almost two weeks. But this is a non-trivial collection of work, so *do not procrastinate*. And finish the parts in order.

ChangeLog

- 13 Mar 23: in instructions for capturing packets on `tap0` with Wireshark, added necessary commands to create and bring up `tap0`

Background

In this assignment (and over the remainder of the course), you will write software that performs analogous operations to both Ethernet network interface controllers (NICs) and network code in the kernel (both NIC drivers and protocol implementations).

Rather than go spelunking in the kernel itself, your code will run as a normal user process and connect to a simulated Ethernet switch. If you followed the FreeBSD installation instructions I wrote, you will have installed the `vde2` package; it contains all the programs you need for the following. (If you didn't install this package, you should.)

The incantation to instantiate a virtual switch is short:

```
$ vde_switch
```

This program must be running whenever you want to send and receive frames. When you want to end it, press Control-C in the terminal where it's running.

Causing a process to connect to this switch and then send or receive frames is a bit more involved, so I have written a couple example programs for you: sender.c and receiver.c. Both of them use functions from `util.h`/`util.c`. They also use functions from cs431vde.h and cs431vde.c; you don't need to understand *how* the functions in these latter files work (though you are welcome to look at them all you like), just how to use them, which is demonstrated in the first two files.

First start `vde_switch` in a terminal.

In a separate terminal, run `receiver`.

In yet another terminal, run `sender`.

You should see the following output in the latter two terminals:

```
$ ./sender
sending frame, length 64
Press Control-C to terminate sender.
^C
```

```
$ ./receiver
received frame, length 64:
ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff
ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

For the time being, always pass `local_vde_cmd` to `connect_to_vde_switch()`; in time, we will use `remote_vde_command` to attach to a virtual switch running elsewhere.

Seeing frames in flight

If you want to use Wireshark to examine the frames you're sending (and it would be good practice here, when the frames are fairly simple, before they get complex later on), you will have to run another few commands to provide an imaginary interface for Wireshark to capture on:

```
$ sudo ifconfig tap0 create
$ sudo ifconfig tap0 up
$ sudo vde_plug2tap tap0
```

The first two commands need to be run once each time the virtual machine is booted. The final command needs to be running as long as you want to capture packets—ie, leave it running while you're sending and capturing; press Control-C to terminate it when you're done working.

If you're using the GUI on the VM, run Wireshark there and select the `tap0` device to capture.

Alternatively, if you're using UTM, from a terminal running on your host, run

```
$ ssh username@192.168.64.2 'tshark -l -i tap0 -w -' | wireshark -k -i -
```

(If tshark isn't installed on your vm: `sudo pkg install tshark`)

Part I: Ethernet for the Host (core)

Write a program called `stack` that simulates the functionality of a network interface (both hardware and in-kernel software). It will connect to a `vde_switch`, receive Ethernet frames from the switch, verify their correctness, and print out a descriptive message for each.

Here is an example output:

```
$ ./stack
received 146-byte broadcast frame from 77 88 99 aa bb cc
received 88-byte frame for me from 88 99 aa bb cc dd
ignoring 1078-byte frame (not for me)
ignoring 96-byte frame (bad fcs: got 0xdeadbeef, expected 0xe88f7195)
ignoring 54-byte frame (short)
```

Your program…

- MUST choose its own Ethernet address, in a way that is easily discernible by reading the code (ie, I should be able to easily discover the Ethernet address you've chosen);
- MUST verify the length and frame-check sequence of the incoming frame;
- MUST print a message for each frame received: if the frame was ignored, the reason MUST also be printed; if the frame was not ignored, its source MUST be printed, as well as whether or not the frame was broadcast;
- MUST accept an arbitrary number of frames (ie, it MUST process frames until the user terminates it with Control-C).

Here is an implementation of the CRC32 algorithm that can be used to calculate the frame check sequence: crc32.c.

You SHOULD probably write a second program that allows you to test this one.

You MAY create any number of source files you like, named whatever you like, but I MUST be able to build your program by running this command in the top-level directory of your course git repository:

```
$ make stack
```

Part II: Binary Exponential Backoff (intermediate)

(Shamelessly stolen from *Computer Networks: A Systems Approach*, by Peterson and Davie.)

Suppose N devices are waiting for another frame to finish on an Ethernet. All transmit at once when the frame is finished and thus collide. Write a program to simulate the devices' attempts to send their frame; its output should be the time slot in which the final device was able to send.

You may make some simplifying assumptions.

- Assume that our unit of time is not seconds, but rather slots (the 51.2 microsecond slice of time we talked about in class).
- Treat a collision as consuming one timeslot (so a collision at time t followed by a backoff of 0 would result in a retransmission attempt at time $t + 1$).
- Assume that there is no time wasted between frames being sent (ie, devices don't wait 9.6 microseconds after sending one frame beore sending another).
- Assume that all frames take one time unit to send.

Otherwise, implement binary exponential backoff as discussed in class.

Your program should take as a single command-line argument the number of devices, N, to simulate. Your program should output a single integer: the time at which the final frame is sent successfully without collision.

Run your program with $N = 10$, $N = 20$, $N = 40$, and $N = 80$. Hypothesize how the delay incurred by collisions scales with N. Summarize your results and your hypothesis in a file called README, which should also contain documentation describing how you obtained the results.

You will need a source of randomness: use `getrandom(2)`, with the `flags` parameter set to 0.

Put all files pertaining to this part in a subdirectory of your course git repository called "ethernet-study".

Part III: Hubs vs Switches (advanced)

Write a program to simulate the behavior of (simplified) Ethernet hubs and switches. Make the same simplifying assumptions as in the previous part.

Your program will simulate random frames being sent between ports on the switch; it will measure only how many frames were unable to be delivered.

During each time slot, every connected device should, with 50% probability, decide to send a frame. If a given device does choose to send a frame, it should randomly choose a device on another port to which to send this frame.

Simulate the behavior of both hubs and switches under these circumstances. You may assume that devices do not attempt to resend in the face of collision and that the switch does not buffer any frames if they are unable to immediately deliver them (ie, the switch silently discards the frame). You may also assume that the switch knows which port each device is connected to before the simulation begins.

Your program should take two command-line arguments: the number of devices connected to the hub/switch and the duration of the simulation to run (an integer number of time slots). It should output the percentage of frames it successfully delivered.

Gather the following data.

- the percentage of frames were successfully delivered by the hub with $N = 10$, $N = 20$, and $N = 40$ devices; and
- the percentage of frames were successfully delivered by the switch with $N = 10$, $N = 20$, and $N = 40$ devices.

In both cases, run your tests 5 times, for 1000 timeslots each, and average the results. Record your results, appropriately labeled, within the README from Part II, along with documentation describing how you obtained the results.

Put all files pertaining to this part in the same directory, "ethernet-study", that you used for Part II.

Style requirements

Beyond the function prototypes as required, none. You are, however, *strongly* encouraged to use good style (comments, appropriate names, etc), as it will benefit you greatly when you need to revisit your code throughout the semester. The idea here is to demonstrate why good style is a good thing, not by forcing you to use it, but by forcing you to deal with the consequences if you don't.

Submission Process

None. I will pull from your git repo at 12pm (noon) on Friday, 31 March and consider that your submission. I will review your code and provide feedback that enumerates improvements required to earn credit, upon receipt of which you will have one week to make revisions.

Assessment

Part I is the core requirement, which you must complete to earn a C.

Part II is the intermediate requirement; you must complete both Parts I and II to earn a B.

Part III is the advanced requirement; you must complete all three parts to earn an A.

Last modified: 03/31/2023 12:56:39