Assignment 4: TCP

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 for you to cement your knowledge of TCP, by writing code that performs one end of its operation.

The three parts of this assignment build on each other, so do not proceed to Part III until you have finished Part II, and do not proceed to Part II until you have finished Part I.

Background and setup

Unlike previous work in this course, in which your programs did not interact with Other People's Code, in this assignment you will use `netcat` as the opposite end of the TCP connection. Since you will run `netcat` on FreeBSD, you will need a network path from `stack` to the FreeBSD kernel itself. Fortunately, you have already done that, in the form of the `tap` device which allowed Wireshark to show you the packets were you sending and receiving.

One thing left a bit implicit in previous discussions of the `tap` device approach is that it is an interface of the FreeBSD kernel, meaning that frames sent to it are handled by the main FreeBSD Ethernet layer. In preceding weeks, you instantiated a `vde_switch` (or several) and sent frames to it/them. If you wanted to observe those frames in Wireshark, you ran `vde_plug2tap`, which caused a `tap` device to be connected to a `vde_switch`, and then pointed Wireshark at the `tap` device. So you might be wondering: why didn't FreeBSD do anything with the frames sent to that `vde_switch` if `vde_plug2tap` sends them on to the `tap` device, which is an ingress point to the FreeBSD kernel?

The reason is that we never assigned MAC or IP addresses to the `tap` device, and therefore FreeBSD had no way of knowing which frames/packets were intended for it. In this assignment, that will change: we will assign MAC and IP addresses to the `tap` device, which will cause a) appropriately-addressed packets sent to the `vde_switch` to be processed by the FreeBSD kernel itself and b) processes running on FreeBSD, using the traditional networking system calls, to cause appropriately-addressed packets to sent to the `vde_switch`. (Okay, I lied a bit: we need to assign an IP, but the MAC is assigned automagically for us. Fortunately, the assigned MAC is consistent.)

Following is a shell script that will perform that setup. It MUST NOT be run as root: otherwise `stack` will not be able to connect to the switch. Remember to `chmod +x` the script to make it executable.

```
#!/bin/sh
#
# Creates and configures a tap device on FreeBSD along with a vde_switch
# connected to it.

network="192.168.0.0"
netmask="255.255.255.0"

vde_switch_ctrl_file="/tmp/net0.vde"
tap_device="tap0"
kernel_endpoint_ip="192.168.0.4"

# Use -hub so Wireshark can see unicast frames.
vde_switch -sock "$vde_switch_ctrl_file" -hub -daemon

sudo ifconfig tap0 destroy
sudo ifconfig tap0 create
sudo ifconfig tap0 "$kernel_endpoint_ip" netmask "$netmask" up

sudo vde_plug2tap --sock "$vde_switch_ctrl_file" "$tap_device" --daemon
```

Part I:

NULL. This part is automatically considered complete.

Part II: single, passive TCP connection

Modify your `stack` program from assignment 3 to accept a single TCP connection: that is, if it receives an appropriately-addressed TCP SYN packet, it should respond with a SYN-ACK and expect a subsequent ACK. After the three-step handshake, any data received over the connection MUST be printed to stdout. In your implementation, `stack` MAY choose not to send data of its own, but rather only acknowledge data it receives. The port upon which `stack` listens for incoming connections MUST be easily configurable at compile time.

You MAY implement the entire TCP state diagram, but there are several corner cases that are not required. You MUST implement the following:

- passive open, traditional three-step handshake (ie, receive SYN, send SYN/ACK, receive ACK);
- receive and acknowledge TCP segments containing data, and write that data to stdout;
- on receiving a FIN segment, respond with an acknowledgement, followed by a FIN segment back; drop the ensuing ACK.

You MAY assume the following:

- packets containing segments will only ever arrive on a single interface;
- segments will never arrive out of order;
- segments will never go missing;
- segments will never contain the URG, PSH, or RST flags.

(Which, yes, I realize this sidesteps much of the interesting host functionality of TCP, but we only have so much time.)

Note that, for this part to work, FreeBSD will need to have an ARP cache entry for the IP address of the `stack` interface connected to the same network. If your `stack` responds to ARP requests, you're set. If not, you can manually add the ARP entry like so:

```
$ sudo arp -s 1.2.3.4 11:22:33:44:55:66
```

Part III: multiplexing TCP connections

Extend your Part II implementation to permit multiple, simultaneous TCP connections. Data received that is printed to stdout MUST be prefixed with text identifying which connection it arrived on.

Additionally, your implementation MUST allow the user to send data *from* `stack` *to* `netcat` by entering text from the keyboard. You will need to implement a way for the user to identify which active connection to send the data over. Finally (as it were), your implementation MUST allow the user to indicate that a particular connection be closed, in which case `stack` must send the appropriate FIN packet.

Your implementation MUST be single-threaded.

You MAY assume that no more than 10 TCP connections will ever be made to `stack`.

Your implementation MAY allow active creation of connections (ie, sending the initial SYN rather than waiting for one) at the behest of the keyboard driver.

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.

I MUST be able to build your program using the following command in the root of your repo:

```
$ gmake stack
```

I MUST be able to run your program using the following command in the root of your repo:

```
$ ./stack
```

Submission Process

Unlike previous assignments in this course, you may request feedback at any point until the final deadline. The final deadline for this assignment is noon on Tuesday, 23 May (the last day of finals period).

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: 05/16/2023 12:01:41