Middle East Technical University

Department of Computer Engineering

CENG 435

Data Communications and Networking

Fall 2022–2023

THE - 2

Due date: 2022-12-05 23:59

1 Introduction

In this assignment, we are going to implement a chat program using UDP Layer 4 protocol talking across an unreliable channel. The assignment will cover *socket programming*. You will use `C` or `C++` to implement it although `C` is recommended.

Before starting the implementation, please read Section 3.4 – *Principles of Reliable Data Transfer* from *Kurose & Ross* to cover the theory. For the practice, you can read Beej's Guide to Network Programming (Using Internet Sockets). Finally, to understand the big picture regarding the reliable data transfer, you can check out the interactive material on the Kurose & Ross website and this animation.

You can discuss the homework and ask your questions on the discussion forum thread on our ODTUClass page. I am also available at yigit@ceng.metu.edu.tr.

2 Setup

Please follow the instructions given in this section thoroughly to set up your homework environment.

We will use an Ubuntu Vagrant box as the host machine and create two network containers in it. By adjusting the parameters of the interfaces (at both ends) between the client and the server, we will have a seamless and native unreliable channel without relying on any external programs.

Run the following commands on an empty folder. If you follow along, you can code on your host machine which is synced with the Vagrant box, so you can compile and run your client and server on there.

```
(host) vagrant init ubuntu/jammy64 # create a new ubuntu 22.04 box
# Places a Vagrantfile on the empty directory
# We will develop alongside it
(host) mkdir code
```

Now we have a directory with a `Vagrantfile` and the `code` directory inside. We can edit the `Vagrantfile` to have the `code` directory synced inside the Vagrant box.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure("2") do |config|
    config.vm.box = "ubuntu/jammy64"
    # config.vm.synced_folder line is commented out, you can edit it
    config.vm.synced_folder "./code", "/home/vagrant/code", type: "rsync"
    # the rest of the file is commented out and is not important
end
```

You can use `vagrant rsync` to *push* your code inside the Vagrant box or `vagrant rsync-auto` on another terminal (in the background) to automate the process.

The following sets up the containers.

```
(host) vagrant up
(host) vagrant ssh
# Set $TERM and useful aliases to enter containers quickly
(vagrant) echo -e "export TERM=xterm-256color" >> .bashrc
(vagrant) echo -e 'alias server="sudo nsenter --net=/var/run/netns/netns0"' >> .bashrc
(vagrant) echo -e 'alias client="sudo nsenter --net=/var/run/netns/netns1"' >> .bashrc
(vagrant) source ~/.bashrc # use the correct TERM and reload .bashrc
(vagrant) sudo apt update && sudo apt upgrade
(vagrant) sudo apt install gcc make

(vagrant) sudo ip netns add netns0
(vagrant) sudo ip link add veth0 type veth peer name ceth0
(vagrant) sudo ip link set veth0 up
(vagrant) sudo ip link set ceth0 netns netns0

(vagrant) sudo ip netns add netns1
(vagrant) sudo ip link add veth1 type veth peer name ceth1
(vagrant) sudo ip link set veth1 up
(vagrant) sudo ip link set ceth1 netns netns1

(vagrant) sudo nsenter --net=/var/run/netns/netns0

(server container) ip link set lo up
(server container) ip link set ceth0 up
(server container) ip addr add 172.24.0.10/16 dev ceth0 # from private IPv4 block
(server container) exit

(vagrant) sudo nsenter --net=/var/run/netns/netns1

(client container) ip link set lo up
(client container) ip link set ceth1 up
(client container) ip addr add 172.24.0.20/16 dev ceth1
(client container) exit

(vagrant) sudo ip link add br0 type bridge
(vagrant) sudo ip link set br0 up
```

```
(vagrant) sudo ip link set veth0 master br0
(vagrant) sudo ip link set veth1 master br0
```

The instructions were adapted from Container Networking is Simple!. If you are curious about the process please refer to that tutorial, but understanding it is not necessary to complete this homework.

You can also follow the steps on this screencast to see what should be happening at each step. At the end of the screencast I demonstrate the `nc` binary as a makeshift server-client chat program. Your binaries will have a similar behavior.

You should run your `client` binary at the client container and the `server` binary at the server container.

2.1 Manipulating Traffic

All this setup for containers would be quite pointless sans this step. Here, we will write `netem` (Network Emulator) rules using `tc` to make the connection between the server and the client *a lot worse*. This section was prepared according to the lab manual from Jed Crandall's course. You can refer to that document for a complete picture.

The following `tc`/`netem` rules are the worst case you will have to deal with. You should run the rules twice inside each container, one for each interface.

```
# on the server, for the server's interface
tc qdisc add dev ceth0 root netem delay 100ms 50ms loss 25% 10% duplicate 25% reorder 25% 50%
# on the client, for the client's interface
tc qdisc add dev ceth1 root netem delay 100ms 50ms loss 25% 10% duplicate 25% reorder 25% 50%
```

While developing, you might want to start with a good channel between the client and the server, and degrade it as you complete the requirements. If you would like to follow that route, you should first start with;

```
# on the server, for the server's interface
tc qdisc add dev ceth0 root netem delay 100ms 10ms
# on the client, for the client's interface
tc qdisc add dev ceth1 root netem delay 100ms 10ms
```

And then change the `tc` rules gradually.

```
# on the server, for the server's interface, note the change rather than add
tc qdisc change dev ceth0 root netem delay 100ms 10ms loss 20%
# on the client, for the client's interface
tc qdisc change dev ceth1 root netem delay 100ms 10ms loss 20%
```

The following are useful as well;

```
# see the current rules in effect
tc qdisc show
# deleting a rule using change is cumbersome
# delete alltogether and start from scratch
# mind the interface
tc qdisc del dev ceth0 root
```

3 Implementation

We are implementing a chat scenario on the command line. Your implementation will output two binaries: a client and a server. The client and the server will only talk to each other, and there will be two participants in the chat. The *messages* will end with a newline character, so every message is a single line. A client-server architecture is ideal for this task. However, there are some constraints.

The network we are on is particularly bad, the packets might be reordered, dropped, delayed or even duplicated. As an additional constraint, the `payload` of any packet you send can have at most 16 bytes. This does include the sequence number or any other metadata you might use to deliver the packet.

The scenario would have been trivial if we were allowed to use `SOCK_STREAM` – kernel's TCP implementation. It would take care of the *reliable transfer* for us. However, we are only allowed to use `SOCK_DGRAM`, to send packets using UDP. You can read more at `$ man 2 socket` on any Linux machine.

You are recommended to start with the interface given in the Section 3.4 of the *Kurose & Ross* book. We are going to implement the **Go-Back-N** protocol, described in the textbook.

To ensure reliable transfer on the unreliable channel, you need to implement `ACK` packets to make sure the other side got your packet right, and a timer mechanism for every packet you sent so that lost packets can trigger a timeout.

The server starts listening on a socket and the client initiates the connection. Then, both programs accept user input from the standard input and send the message to the other side. Any input (on both server and client) will be sent to the other endpoint using reliable data transfer, on UDP packets. The receiving party should see the messages in the same order as they were written by the sending party, with duplicates silently ignored.

Ideally, threading should be used to listen and send at the same time. This is especially important for receiving ACK packets and data packets concurrently, on both endpoints.

To terminate the connection, at any time but not as the first input/message, either server's user or the client's user can enter three consecutive newline characters on `stdin` (two empty lines). This will initiate the shutdown & teardown sequence for both the client and the server. Assume the user sitting on the client input two empty lines. The client sends the shutdown request to the server, still following the reliable transfer protocol. However, no more input from `stdin` will be accepted (e.g. they will be ignored) nor any remaining messages from server will be displayed. When the server receives the shutdown request, it will also ignore the remaining messages in transit and ignore its `stdin`. When either endpoint complete the shutdown process (either through ACKs or sufficient timeouts), they can gracefully exit. The process mirrors for the server as well.

3.1 Notes

Please comment your code thoroughly, on *what* your implementation does rather than *how*. Leaving comments to explain your thought process and choices will help me as well as you when you return to work on your homework the following day.

You can use `printf` to debug and trace your code. One suggestion is to be verbose during the startup/shutdown process. However, please use `stdout` for message content and use `stderr` like `fprintf(stderr, "Inside foo()!");` to keep `stdout` clean and leave your code scriptable (for automatic grading). I will read your code in its entirety as well.

4 Usage

Your submissions will be compiled on `gcc 11.3.0` on Linux.

Your implementation should be compiled with `make`. Here is a tutorial to get you started: https://cs.colby.edu/maxwell/courses/tutorials/maketutor/.

Your implementation should compile into 2 binaries: `server` and `client`.

Please provide any command line arguments your binaries take in a `README.md` file. For instance; `./server <server-port-number>`, `./client <server-ip-address> <server-port-number>`.

However, since both the `server` and the `client` binary will `bind` to their respective containers' IP addresses, there should not be a need to deviate from the usage given above.

5 Submission

This is an individual assignment. Discussing high level ideas regarding your implementation is encouraged. You can build on top of the code examples given in the Linux man pages and external resources such as "Beej's Guide" as long as you indicate their origin through code comments. However, using implementation specific code that is not your own is strictly forbidden and constitutes as cheating. This includes but not limited to friends, previous homework, CENG homework repositories on GitHub, or the Internet in general. The violators will get no grade from this assignment and will be punished according to the department regulations.

Archive your submission as a `.tar.gz` file and name it using your name with underscore characters for spaces. Upload `name_surname.tar.gz` to ODTUClass.

6 Grading

Your submissions will be graded primarily on the correctness of the Go-Back-N protocol, and it's implementation & usage at the server and the client. A lower portion of the grade is reserved for the code comments. Aside from the explicit grade for the code, keep in mind that without comments I might miss your implementation details and grade erroneously.