Middle East Technical University　　Department of Computer Engineering

CENG 435

Data Communications and Networking
Fall 2021–2022
THE - 2

Due date: 2021-12-22 23:59

1 Introduction

In this assignment, we are going to implement a two-user chat program. 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) at `http://beej.us/guide/bgnet/`. 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: `https://www2.tkn.tu-berlin.de/teaching/rn/animations/gbn_sr/`.

You can direct your questions to me at yigit@ceng.metu.edu.tr or to the discussion forum thread on ODTUClass.

2 Implementation

We are going to implement a chat program. The program should work for 2 endpoints, not like a group chat but like a private messaging application. 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, they might be dropped, delayed, or garbled. As an additional constraint, the `payload` of any packet you send can have at most 8 bytes. This does not include the checksum, sequence number, or any other metadata you might use to deliver the packet, but the actual data space is limited.

This wouldn't be so bad if we were allowed to use `SOCK_STREAM` - kernel's TCP implementation, which would take care of the *reliable transfer* for us. However, we are only allowed to use `SOCK_DGRAM`, to send packets with UDP. You can read more at `$ man 2 socket` on any Linux machine.

Selective Repeat (SR) is a pipelined reliable data transfer protocol. If allows maximum channel utilization by retransmitting only the packets that are definitely lost or garbled along the way and nothing else. This utilization comes with a trade-off, selective repeat is not straightforward to implement.

You are recommended to start with the interface given in the Section 3.4 of the Kurose & Ross book. The functions we are interested in are `rdt_send`, `rdt_receive`, `extract_data` and `make_packet`. You do not have to use these functions exactly but it's a good starting point.

To ensure reliable transfer on the unreliable channel, you need to implement checksums for every packet you put on the wire, `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. Selective repeat also requires a window size. You can determine this value yourself by looking at the network parameters.

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.

To emulate an unreliable channel, we are going to use the `troll` binary from CSE-3461 of Ohio State University. You can find the manpage of the `troll` as well as accounts from the students who had taken this course before. `troll` is a simple program, it forwards packets (datagrams) from one socket to another. While forwarding, it can delay, reorder and garble the packets. The course page has additional resources for understanding how the binary works. `troll` expects the client to `bind()` the sending port as well, which is not common (UDP is low-effort, who cares which port you are sending from?).

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

There are 3 major problems you will have to solve. The first is establishing a bidirectional communication channel using UDP packets. The second is discerning between the ACK and the data packets. And the third (which is *majorer* than the other two) is bringing everything together to work with the selective repeat protocol.

2.1 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.

To test your implementation on your own machine, use `127.0.0.1` as the IP number. `localhost` might give you a `Connection refused` error, it's more reliable to go with the `127.0.0.1`.

`troll` is a 32-bit binary. While trying to run `troll`, you might see the following error message:

```
Failed to execute process './troll'.
Reason:  The file './troll' does not exist or could not be executed.
```

You will get this error if you Linux distribution does not come with 32-bit libraries out of the box, you should install them manually. The libraries can be found for your particular distro by the keywords: i386, multilib, 32-bit gcc etc.

To test your implementation on your own machine, you can use the following to start up the `troll`.

- First: `./troll -C 127.0.0.1 -S 127.0.0.1 -a 5202 -b 5201 5200`
- Second: `./server 5202`
- Third: `./client 127.0.0.1 5200 5201`

This will make the `client` bind to port `5201`, `server` to listen at `5202`, and `troll` to listen at `5200` to forward from client's port to server's port.

You can use `printf` to debug and trace your code. 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 go over your code manually as well.

2.2 Bonus

If you would like to challenge yourself, you can optimize your implementation. There's a `benchmark.sh` provided on ODTUClass. I will give a small prize to the *fastest* implementation on that benchmark (on my machine to keep things fair), that complies with all the restrictions given above.

3 Usage

Your submissions will be compiled on `gcc 11.1.0` on Linux. You can use `inek's` for testing.

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 <client-port-number> <server-port-number>`. Your binaries should work with the `troll` in the middle, where the client sends to and the server receives from `troll`, or without it where client and server are connected to each other.

While testing your code, you can try different `troll` variables to ensure your implementation works under different conditions. The flags to look for are: `-g <percentage>` to garble a percentage of packets, `-s <delay>` and `-se <delay>` for the delay of the packets between two endpoints, `-x <percent>` to drop some packets on the wire and `-m <percent>` to duplicate packets by a percentage. `-r` flag prevents reordering of the packets on the wire, some test cases will not use this flag.

4 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 homeworks, 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.

5 Grading

Your submissions will be graded primarily on the correctness of the selective repeat protocol and it's implementation & usage at the server and the client. The much smaller portion of the points will be awarded based on the comments on your code.