**Exercise 3: Uplink Established**

This exercise is part of a three-stage process:

* This first exercise is to make you more familiar with using TCP and UDP for communication between processes running on different machines. Do not think too much about code quality here, as the main goal is familiarization with the protocols.
* Exercise 4 will have you consider the things you have learned about these two protocols, and implement (or find) a network module that you can use in the project. The communication that is necessary for your design should reflect your choice of protocol. This is when you should start thinking more about code quality, because ...
* Finally, you should be able to use this module as part of your project. Since predicting the future is notoriously difficult, you may find you need to change some functionality. But if the module has well-defined boundaries (a set of functions, communication channels, or others), you *should* be able to swap out the entire thing if necessary!

Note:

* You are free to choose any language. Using the same language on the network exercises and the project is recommended, but not required. If you are still in the process of deciding, use this exercise as a language case study.
* Exactly how you do communication for the project is up to you, so if you want to venture out into the land of libraries you should make sure that the library satisfies all your requirements. You should also check the license.

Practical tips:

* Sharing a socket between threads should not be a problem, although reading from a socket in two threads will probably mean that only one of the threads get the message. If you are using blocking sockets, you could create a "receiving"-thread for each socket. Alternatively, you can use socket sets and the [select()](http://en.wikipedia.org/wiki/Select_%28Unix%29) function (or its equivalent).
* Be nice to the network: Put some amount of sleep() or equivalent in the loops that send messages. The network at the lab will be shut off if IT finds a DDOS-esque level of traffic. Yes, this has happened before. Several times.
* You can find [some pseudocode here](https://github.com/TTK4145-students-2019/exercise-3-ttk4145_group14/blob/master/resources.md) to get you started.

**Part 1: UDP**

We have set up a server on the real time lab that you're going to communicate with in this exercise. If you cannot connect to it, it might be down. Ask a student assistant to turn it on for you.

**Receiving UDP packets, and finding the server IP:**

The server broadcasts it's own ip on port 30000, listen for messages on this port to find the IP. You should write down the IP address as you will need it for again later in the exercise.

**Sending UDP packets:**

The server will respond to any message you send to it. Try sending a message to the server ip on port 20000 + n where n is the number of the workspace you're sitting at. Listen to messages from the server and print them to a terminal to confirm that the messages are in fact beeing responded to.

* The server will act the same way if you send a broadcast (#.#.#.255 or 255.255.255.255) instead of sending directly to the server.
  + If you use broadcast, the messages will loop back to you! The server prefixes its reply with "You said: ", so you can tell if you are getting a reply from the server or if you are just listening to yourself.
* You are free to mess with the people around you by using the same port as them, but they may not appreciate it.
* It may be easier to use two sockets: one for sending and one for receiving. You might also find it is easier if these two are separated into their own threads.

**Part 2: TCP**

There are three common ways to format a message: (Though there are probably others)

* 1: Always send fixed-sized messages
* 2: Send the message size with each message (as part of a fixed-size header)
* 3: Use some kind of marker to signify the end of a message

The TCP server can send you two of these:

* Fixed size messages of size 1024, if you connect to port 34933
* Delimited messages that use \0 as the marker, if you connect to port 33546

The server will read until it encounters the first \0, regardless. Strings in most programming languages are already null-terminated, but in case they aren't you will have to append your own null character.

**Connecting:**

The IP address of the TCP server will be the same as the address the UDP server as spammed out on port 30000. Connect to the TCP server, using port 34933 for fixed-size messages, or port 33546 for 0-terminated messages. It will reply back with a welcome-message when you connect.

The server will then echo anything you say to it back to you (as long as your message ends with '\0'). Try sending and receiving a few messages.

**Accepting connections:**

Tell the server to connect back to you, by sending a message of the form Connect to: #.#.#.#:#\0 (IP of your machine and port you are listening to). You can find your own address by running ifconfig in the terminal, the first three bytes should be the same as the server's IP.

This new connection will behave the same way on the server-side, so you can send messages and receive echoes in the same way as before. You can even have it "Connect to" recursively (but please be nice... And no, the server will refuse requests to connect to itself).