Introduction

Your objective is to write a client-server application which implements a network speed test. The app will allow you to compare UDP and TCP downloads and see how they share the same network.

Each team in the course will write both a client application and a server application, and everybody's clients and servers are expected to work together with full compatibility.
Please include plenty of comments in the code to improve readability.

Example Run

1. Team Mystic starts their server. The server prints out "`Server started, listening on IP address 172.1.0.4`" and starts automatically sending out "offer" announcements via UDP broadcast once every second.
2. Team Valor starts their server, which prints out "`Server started, listening on IP address 172.1.0.88`" and also starts automatically sending out "offer" announcements over UDP once every second.
3. Team Instinct starts their client. The client asks the user for the file size, the amount of TCP connections and the amount of UDP connections. The user specifies a 1GB download, 1 TCP connection and 2 UDP connections.
4. Teams Rocket, Beitar and Katamon similarly start their clients. The clients all print out "`Client started, listening for offer requests...`".
5. All of the clients get the Mystic announcement first, and print out ""`Received offer from 172.1.0.4`"
6. The Team Instinct client connects to the Team Mystic server over TCP. After the connection succeeds the client sends the file size over TCP connection, followed by a line break ('\n'). The Team Instinct client also connects using UDP by sending a request packet including the requested file size. The client should log the start time of the transfer so it can measure the transfer rate once the transfer finished. All requests are sent in parallel - please use threads.
7. All other clients similarly send their requests to the server.
8. The server responds to the TCP request by sending the requested amount of bytes over the TCP connection and then hanging up. The server responds to the UDP request by sending multiple UDP packets, each with an included sequence number, containing parts of the data. The client detects that the UDP transfer concludes after no data has been received for 1 second.
9. For each connection, the client should measure the time it takes for the file to be transferred, and (for UDP), the percentage of packets which were discarded on the way. At the end of each transfer, the client should print out the following for TCP: "`TCP transfer #1 finished, total time: 3.55 seconds, total speed: 5.4 bits/second`", and the following for UDP: "`UDP transfer #2 finished, total time: 3.55 seconds, total speed: 5.4 bits/second, percentage of packets received successfully: 95%`".
10. When all transfers are completed the client should print out: "`All transfers`

`complete, listening to offer requests”` and immediately return to step 4.

Suggested Architecture

The client is a multi-threaded app, which has three states:

- Startup. You leave this state when you're done asking the user for parameters
- Looking for a server. You leave this state when you get an offer message.
- Speed test. When you enter this state you should launch multiple threads, one for each TCP and UDP connection. You leave this state when all file transfers are finished.

The server is also a multi-threaded app - it constantly sends offer messages in one thread, and launches a new thread to handle each incoming speed test request (either UDP or TCP).

Packet Formats

There are three UDP packet types: offer (server to client), request (client to server) and payload (server to client).

This is the format of the **offer** message:

- Magic cookie (4 bytes): 0xabcddcba. The message is rejected if it doesn't start with this cookie
- Message type (1 byte): 0x2 for offer (server to client)
- Server UDP port (2 bytes): The port on the server that the client is supposed to connect for UDP requests.
- Server TCP port (2 bytes): The port on the server that the client is supposed to connect for TCP requests.

This is the format of the **request** message:

- Magic cookie (4 bytes): 0xabcddcba. The message is rejected if it doesn't start with this cookie
- Message type (1 byte): 0x3 for request
- File size (8 bytes): The amount of data requested for transfer, in bytes.

This is the format of the **payload** message:

- Magic cookie (4 bytes): 0xabcddcba. The message is rejected if it doesn't start with this cookie
- Message type (1 byte): 0x4 for payload
- Total segment count (8 bytes): The total amount of segments in the entire data stream.
- Current segment count (8 bytes): Identifies the current segment number.
- Everything else - actual payload. You should measure how many bytes are in this part but you can otherwise ignore it.

For the TCP transfer, the client sends the amount of data requested for transfer, in bytes, as a regular string, followed by a new line ("\n"), and the server responds with the entire file.

Tips and Guidelines

- Please pick a creative name for your team - there will be a contest.
- Both server and client applications are supposed to run forever, until you quit them manually.
- It's best to do the development on your own personal phone hotspot. During testing all of the teaching staff (Yossi, Nadav, and Ron) will be sharing a testing network. Be prepared to experience and tolerate interference from other teams being tested at the same time!
- The server does not have to listen on any particular port, since this is part of the offer message.
- **Do not use busy-waiting (e.g. while-continue loops).** As a general guideline, if your program consumes more than 1% CPU on your own computers, you're probably doing something wrong.
- Think about what you should do if things fail - messages are corrupted, the server does not respond, the clients hang up, etc. Your error handling code will be tested.
- If you try to run two clients on the same computer, they won't be able to listen on the same UDP port unless you set the SO_REUSEPORT option when you open the socket, like this:

```
>>> s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

>>> s.bind(('',13117))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 98] Address already in use

>>> s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
>>> s.bind(('',13117))
```

- The assignment will be written in Python 3.x. You can use standard sockets or the scapy package for network programming. The struct.pack functions can help you encode and decode data from the UDP message.
- Please set up a git repository for your own code and make sure to commit regularly, since the file system on the hackathon computers is unstable. Visual Studio Code has git support so it's quite easy.

To Get Full Points on Your Assignment

The grading will be competitive, and only the top 10% of the teams will receive grades above 90. We will publish more detailed excellence criteria later, but here are some times to get you started:

- Work with any client and any server
- Write high-quality code (see below)
- Have proper error handling for invalid inputs from the user and from the network, including timeouts
- Use ANSI color to make your output fun to read
- Collect and print interesting statistics

Code Quality Expectations

How to Submit

Please submit to the Moodle a link to your github repository (make sure it's not private). You can commit as much as you want, but only the last commit before the deadline will be considered.

Static Quality

- Code has proper layout, and meaningful function and variable names
- Code has comments and documentation for functions and major code blocks
- No hard-coded constants inside code, especially IP addresses or ports

Dynamic Quality

- Return values of functions are checked
- Exceptions are handled properly
- No busy-waiting!
- Network code is isolated to a single network

Source control

- Code is hosted in a github repository
- Commits were made by all members of the team
- Proper use of commit messages and branches