CS105 – Computer Systems Fall 2024

Lab 10: Networking Lab

Due: Wednesday, December 4, 2024 at 11:59pm

The goal of this activity is to get some experience with TCP sockets and network programming. And to have some fun! :-) In this activity, you will implement a simple client-server messaging system that will send messages back and forth over the Internet. You may complete this activity with a partner or in a small group. **This is an in-lab activity. Submit whatever you have completed at the end of lab (8:15pm). You will get full credit as long as you participated, even if your code is not finished or not working correctly.**

As usual, the starter code is available as a tar file `netlab.tar` that can be downloaded from the course website or can be found in the `/cs105` directory on the course VM. To get started, unpack the tar file using the command (e.g., `tar xvf /cs105/starters/netlab.tar`), and look inside. You should see five files: `imclient.c`, `imclient.py`, `imserver.c`, `imserver.py`, and `Makefile`. This activity can be completed in either C or in Python. Before you go any further, you should decide which language you want to use to complete the assignment. If you choose to solve the assignment in C, you can use the Makefile to compile your code. But honestly, I find socket programming (much) easier in Python and strongly recommend doing this assignment in Python.

It will probably be easier to work on your own computer than on the VM: only one process can be bound to a given port at a time, and working on your own computer will reduce contention. However, you may want to run the server code on the VM once you get it working just to see it actually work!

1 Your Tasks

Your tasks are to implement a server and a client that can exchange messages over the network. You will use TCP sockets to implement this task; a summary of the TCP socket lifecycle, along with descriptions of the relevant functions in both C and Python, is given in Appendix A.

Server. The server will take one commandline argument: a port number. When the server starts up, it will immediately start listening for incoming connections by opening a socket, binding the socket to that port (and the computer's IP address), and listening for incoming connections. Once it accepts an incoming connection, it will accept that connection and then enter an infinite loop in which it receives a message from the client, prints that message to `stdout`, then reads a line from `stdin` and sends that response to the client.

Client. The client will take two commandline arguments: an IP address (the IP address of the computer the server code is running on) and a port number (the port number the server is using). The client will open a socket and initiate the TCP connection with the server. Once a connection has been established, the client will enter an infinite loop in which it reads a line from `stdin` and send that to the server, then waits for a response from the server and prints that response to `stdout`.

A sample interaction between client and server shown below:

```
$ ./client 127.0.0.1 8080
Enter message for server: hello1
Received from server: hello2
Enter message for server:
```

Client

```
$ ./server 8080
Received from client: hello1
Enter message for client: hello2
```

Server

2 Feedback

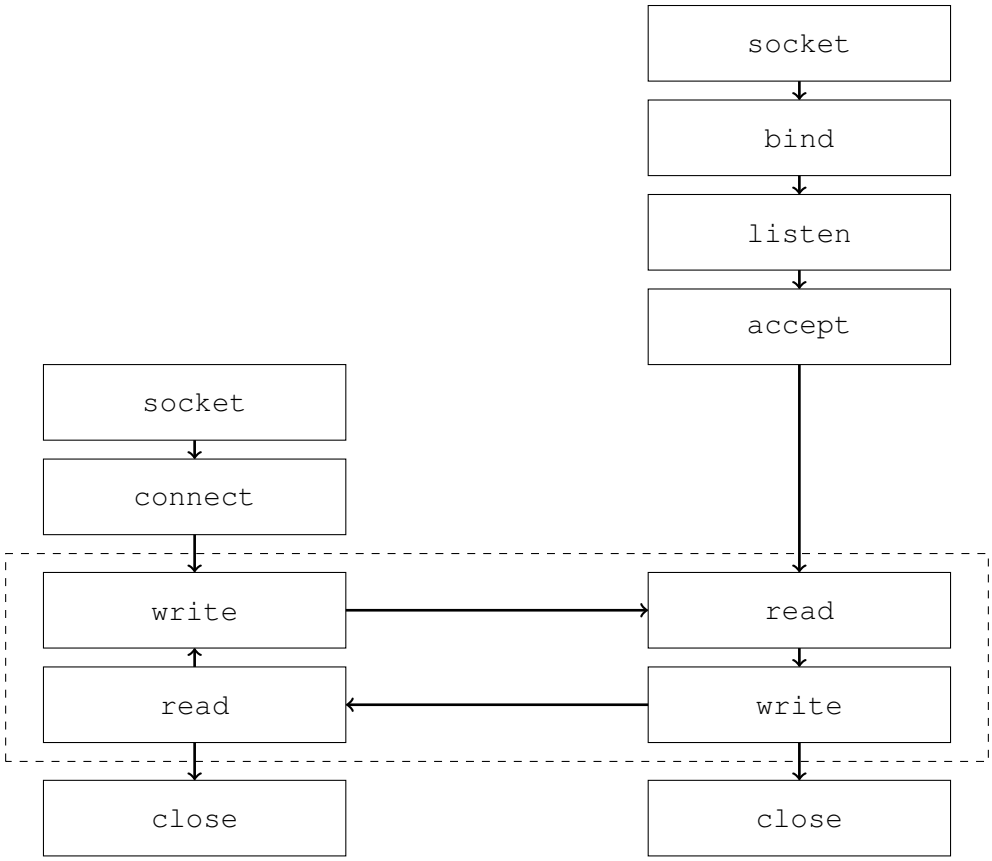
Create a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this activity?
2. Were you able to successfully complete the full actiity?
3. Any comments on this lab activity?

As always, how you answer these questions **will not affect your grade**, but whether you answer them will.

A TCP Sockets

The socket interface is a standard tool for implementing low-level networked programs. Programs that work with TCP sockets all follow the same pattern. First the server starts up and opens a socket; it then binds this socket to its IP address (or one of its IP addresses, if the computer has multiple) and to some specified port number. Many of the lower port numbers are reserved for particular types of applications (and might be in use on your machine!), so I'd recommend that you bind to a port number above 8000. The server then listens for incoming connections. Accepting an incoming connection opens a new file descriptor, which is then used to read and write over the TCP connection. When the client starts up, it opens a socket and connect to the server using the server IP address and the port associated with the listening socket. It then reads and writes to the server using this socket. This design pattern is summarized in the diagram below.



Note: the following descriptions are not complete, they are just intended to give you some hints for getting started. You will probably find yourself needing to look at the socket documentation for the language you are using.

A.1 Opening a Socket

To open a socket, you need to specify the type of IP address you will use and the type of socket you will use. The value for IPv4 addresses is `AF_INET` and the value for TCP sockets is `SOCK_STREAM}`. In C, this looks like

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
```

In Python, it looks like

```
sock_fd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Hint: Python is not good about closing sockets and freeing up ports, so if you are solving this assignment in Python, you should configure your socket for reuse immediately after opening it using the following command:

```
sock_fd.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

A.2 Binding a Socket

Before you can use a socket, you need to bind it to an IP address and a port. It is good practice to not hardcode the IP address of your computer (as this will likely change!) and instead use an IP address currently assigned to that computer (in C you construct `INET_ANY`, in Python you will use an empty string). The signature for the bind function in C is

```
int bind(int sockfd, const struct sockaddr * addr, socklen_t addrlen);
```

In Python, it is

```
socket.bind(address)
```

where `address` is an IP address-port tuple.

Hint: Networking tends to use big-endian numbers. If you are solving this assignment in C, you will need to explicitly convert your IP addresses/ports into network order. You might want to investigate the functions `htons`, `htonl`, and `inet_addr`.

A.3 Listening for a Connection

In C, the signature for the listen function is

```
int listen(int sockfd, int backlog);
```

In Python, it is

```
socket.listen([backlog])
```

A.4 Accepting a Connection

In C, the signature for the accept function is

```
int accept(int sockfd, struct sockaddr * addr, socklent_t * addrlen);
```

In Python, it is

```
socket.accept()
```

A.5 Initiating a Connection

The client initiates a TCP connection using the connect function. In C, the signature for this function is

```
int connect(int sockfd, const struct sockaddr * addr, socklen_t addrlen);
```

In Python it is,

```
socket.connect(address)
```

where `address` is a IP address-port tuple.

A.6 Reading and Writing to a Socket

Given a connected TCP socket file descriptor, you can read and write to the network just the same as you would read and write to a file! In C, the UNIX IO functions are `read` and `write`. In Python, you will want to use the `send` and `recv` methods.

Hint: The Python methods expect sequences of bytes, not strings. You might want to investigate the `encode` and `decode` methods.

A.7 Closing a Socket

It is good form to close your socket when your done. However, since your code involves infinite loops, in this case you can get away with skipping this step.