CS 214: Introduction to computing systems
Winter Term, 2026

# Lab 9

## Networking with sockets

In this lab, you will learn how to make programs communicate across a network. We will be using *sockets* for this communication. A socket is an abstraction for the endpoint of communication. This lab uses a traditional approach to sockets; modern code would likely use other libraries for additional functionality (like encrypted connections), but I believe these are still built on top of the sockets abstraction.

Begin by logging onto euclid and copying the files from `/home/courses/cs214/lab9` using the following:

```
cp -R /home/courses/cs214/lab9 .
```

This copies the `lab9` subdirectory into wherever you call it from. You'll need to `cd lab9` into it. The directory contains five files from "TCP/IP sockets in C" by Donahoo and Calvert. (I rearranged some of their code to simplify it for our lab.)

These files are the source code for two programs. The first program is a *server*, a program that runs and waits for another program to make a request. In this case, the server simply repeats the message it receives back to the sender. The second program, the *client*, connects to the server, sends a message and then prints the reply. The main files for these two programs are `TCPEchoServer4.c` and `TCPEchoClient4.c`, respectively. The file `DieWithMessage.c` provides functions to print an error message and the file `Practical.h` has headers for functions in the other files.

Compile these programs with the command line

```
gcc -std=c99 -Wall -o client TCPEchoClient4.c DieWithMessage.c
gcc -std=c99 -Wall -o server TCPEchoServer4.c DieWithMessage.c
```

(Note that multiple C files can be compiled into a single executable.) You can also use a compilation tool called `make` which uses data in the file `makefile` to generate these lines itself:

```
make
```

This is an older style of build system.

Look through these programs to see how sockets are created and used. The server creates the socket, binds it to a specific port, and then listens for connections. (Messages to a computer have a specific destination port, which tells the operating system which program to deliver that message to. If you think about messages on the network as mail being sent, a *port* is like an apartment number; the message has already found the right building, but now needs to be delivered to the right apartment.) Then it accepts the connection, which generates another socket, one specifically for this communication. Once the connection is established, it uses `send` and `recv` to handle messages on that connection. The client also creates a socket, but it just needs to connect to the server before using `send` and `recv`. Both programs close the connection once they are done with it. Another important difference between the server and client is that the client needs not just the port number, but also the address of the computer running the server (the building address in our analogy). The address is an *IP address*.

Once you've looked through the programs a bit, it is time to run them. To start the server, the command is

`./server` *port_number*

where you can give any unused integer at least 1024 as *port_number*. (Low-numbered ports are reserved for the operating system.)

Then open another window and run the client with

```
./client 127.0.0.1 hello  port_number
```

where the port number must be the same as the number used when starting the server. This causes the client to send the server the message "hello", which is echoed back by the server and then printed by the client. The 127.0.0.1 is a special IP address that always refers to the local machine.

Now see if you can modify the code to write a simple communication program something like instant messaging (but much more primitive). You'll want to include the behavior of both the client and the server in the combined program.

One of the programs's first tasks is to figure out whether it is the client or server. This is done via the arguments it receives on the command line (when you type the program's name into the command prompt). Your program should take either a port number or a port number and an IP address (the 127.1.1.1 part) from the command line. To receive these arguments and distinguish between the cases, your `main` should have the following signature:

```
int main(int argc, char** argv)
```

This has always been the signature of `main`, but we've been allowed to abbreviate it when we didn't care about arguments. The value of `argc` tells a program how many arguments it received; the value is always at least 1 since the first "argument" is the name of the program. Thus, `argc` will be either 2 or 3 when your program is run correctly. `argv` is an array of strings giving the text of the arguments.

If your program gets just a port number, it acts as the server, listening on the given port. (Base this part on the server code.) If your program gets both a port number and IP address, it acts like the client, opening a connection to the given port on the given machine. (Using code based on the client code...) Once a connection is established, the programs alternate sending each other a line of text. First the client reads a line from standard input and sends it to the server, then the server reads a line from standard input and sends it to the client, etc. To read a line of text, use the command

```
fgets(line, 100, stdin);
```

where `line` is a character array of length 100. (The 100 gives the `fgets` the maximum length of line you'll accept and `stdin` tells it to read from *standard input*, the keyboard where you type.) Use a 0 (the number, not the character) to denote the end of a message. As each line arrives, the recipient program prints it. Either program should close the socket and exit at the end of input (i.e. if you hit control-D; `fgets` will return `NULL` when you do this). Test your program first by communicating with itself and then talking to the program of someone else in the class.