Lab 4: Socket Programming: netcat_part

Overview

The goal of this lab is to familiarize yourself with application level programming with sockets, specifically stream or TCP sockets, by implementing a client/server socket application, netcat_part. Additionally, this lab will introduce you to advanced file manipulation in C using file pointers and file streams. This lab is the first in a sequence that result in a small implementation of a BitTorrent client.

The programming in this assignment will be in C requiring standard sockets, and thus you **will not** need to have root access. You should be able to develop your code on any CS lab machine or your personal machine, but all code will be tested on the CS lab.

Deliverables

Your submission should minimally include the following programs and files:

- netcat_part.c
- Makefile
- README

Your README file should contain a short header containing your name, username, and the assignment title. The README should additionally contain a short description of your code, tasks accomplished, and how to compile, execute, and interpret the output of your programs. Additionally, if there are any short answer questions in this lab write-up, you should provide well marked answers in the README, as well as indicate that you've completed any of the extra credit (so that I don't forget to grade it).

Submission Instructions

To submit this lab you will use handin43. You can retrieve relevant files via update43.

netcat_part partial file transfers

In this lab you will familiarize yourself with basic socket programming and C file I/O primitives by implementing a form of `netcat` for partial file transferring, so dubbed `netcat_part`. First, a description of standard `netcat` is provided followed by a description of the lab requirements for implementing `netcat_part`.

Basics of netcat

The `netcat` program (or `nc` on BSD machines) is a simple networking program that connects to a remote server and sends input from `stdin` over the network to the remote server. It can also function as a server by opening a socket and listening for incoming connection, writing all received data to `stdout`. This is why it is called `netcat`, or the "network version of `cat`". Here is an example of some of its usages:

- `netcat` **as a client**:

  The most common usage of `netcat` is as a client; that is, as a program that connects to a remote server. Here is a standard example:

  ```
  #> echo "GET /˜aviv/ " | netcat web.cs.swarthmore.edu 80
  ```

  This command will connect to the CS web server on port 80 and issue a standard GET request for my web page, i.e `www.cs.swarthmore.edu/˜aviv/`. The CS web server will respond with the HTML of the web page, which will be printed to `stdout` by `netcat`. This example makes use of a shell pipe "|" which directs the standard output of one program as the standard input of another, i.e., `echo`'s standard output of "GET ..." is `netcat`'s standard input.

- `netcat` **as a server**:

  `netcat` also functions as a simple server that will listen for a connection and write all received data to standard out. Consider the command below:

  ```
  #> netcat -p 6767 -l
  ```

  Here, `netcat` is provided two command line arguments: `-p`, which indicates what port to open the socket on; and `-l`, which indicates that `netcat` should listen for incoming connections. You can test your `netcat` server by trying to connect to it using another instance of `netcat`. In another terminal on the same host, issue the following command:

  ```
  #> echo "Hello World" | netcat localhost 6767
  ```

  In the terminal where the `netcat` server is running, you should see "Hello World" printed to the terminal output. Once the connection is closed by the `netcat` client, the `netcat` server will also close its connection and exit.

This is the basic functionality of `netcat`, and the more you use it, the more you'll see how useful it is, particularly as a debugging tool for network applications (*hint!*). For a full description of `netcat`'s functionality, refer to the manual page.

Basics of netcat_part

You and your lab partner, will implement a form of netcat for transferring parts of files. After, running update43, under directory labs/04 you'll find skeleton code netcat_part.c and a usage function, whose output is duplicated below:

```
netcat_part [OPTIONS]  dest_ip file
         -h                      Print this help screen
         -v                      Verbose output
         -p port                 Set the port to connect on (dflt: 6767)
         -n bytes                Number of bytes to send, defaults whole file
         -o offset               Offset into file to start sending
         -l                      Listen on port instead of connecting and write output to file
                                 and dest_ip refers to which ip to bind to (dflt: localhost)
```

Your task is to implement all the functionality in the help reference above. Below, is detailed descriptions and example usages.

- netcat_part **as client**

  Using netcat_part as a client is very similar to netcat, except that instead of reading input from stdin, input is provided in the form of a file, file. For example, here is the same netcat_part code for getting my homepage:

  ```
  #> echo "GET /~aviv/ " > GET_req.txt
  #> netcat_part -p 80 web.cs.swarthmore.edu GET_req.txt
  ```

  First, the GET request is written to a file GET_req.txt using output redirection (if you are unfamiliar with shell I/O redirection, there are many good references online). Next, the netcat_part command is issued such that it connects to the CS web server on port 80 using the input file GET_req.txt as the data to send.

  In addition to this standard usage, netcat_part has the additional feature of being able to send parts of a file, the _part of netcat_part. Consider a large file, such as the text of *Moby Dick*, and you only want to send the first 100 bytes of the file to a remote server rather than the whole file. You can use this netcat_part command:

  ```
  #> netcat_part -n 100 remote.server.com moby_dick.txt
  ```

  And if you want to send the first 100 bytes offset into the file, you can use this similar command:

  ```
  #> netcat_part -o 256 -n 100 remote.server.com moby_dick.txt
  ```

  Which will send 100 bytes of the file starting at byte 256. That is, it will send bytes 256 through 355 of the file.

- netcat_part as server

  When functioning as a server, netcat_part acts nearly the same as netcat except that information sent over the line is written to the file rather than stdout. For example:

  ```
  #> netcat_part -l localhost output_file
  ```

  will open a socket for listening on the localhost and write data to the output file. The user should also be able to set a port for listening:

```
#> netcat_part -p 1024 -l localhost output_file
```

Your version of `netcat_part` functioning as a server does not need to worry about the offset or the number of bytes it receives when writing to the output file. It can just open the output file for writing, truncating the file if it already exists, and begin writing data to the start of the output file.

EXTRA CREDIT 7 points: Add in additional functionality such that the offset and the number of bytes are considered on the server end. For example, consider the offset and byte example above: If you implement the extra credit, then `netcat_part` will write the 100 bytes to the output file at the appropriate offset, and,on successive runs of `netcat_part` for the same output file, it will not truncated the file and continue to write data to the appropriate location in the file. Add a options flag, `-m` for "maintain", that will enable this functionality. Be sure to update the `usage()` and `parse_args()` functions.

(*Hint: you will likely need to exchange some preliminary info between client and server to enable this functionality, such as the the offset into the file among other information.)*

File Programming Preliminaries and Socket Programming

Below are descriptions of the requisite C functions you will need to complete this assignment. First, file I/O is discussed, particularly file streams, and following, the basics of socket programming is discussed, including C-stlyle pseudo-code examples.

File Streams

- **Opening a file**: You are probably already familiar with the basic file opening procededure `open()`, which returns a file descriptor, an `int`. Additionally, there are other ways for manipulating files in C using a file pointer. Consider `fopen()` below:

  ```
  FILE * fopen(const car * filename, const car * restrict mode)
  ```

  `fopen()` opens a file named `filename` with the appropriate `mode` (e.g., "r" for reading, "w" for writing, "r+" for reading and writing, and etc.). The important part to consider is that a file pointer is returned rather than file descriptor. A file pointer allows you to interpret files as streams of bits with a read head pointed to some part of the file.

  For example, if you open a file for reading, the read head will be pointed to the beginning of the file, and if you open a file for append, the read head is pointed to the end of the file. Note that this is the way that Python handles file I/O, so you are already familiar with this form even if you weren't aware.

- **Reading and Writing**: Reading and writing from a file pointer is very similar to that of a file descriptor:

  ```
  size_t fread (void * ptr, size_t size, size_t nitems, FILE * stream);
  size_t fwrite(void * ptr, size_t size, size_t nitems, FILE * stream);
  ```

  Like before data is read from or written to a buffer, `ptr`, but the amount is described as `nitems` each of `size` length. This is very useful when reading chunks of data, but you can always set `size` to

1 and `nitems` to the number of bytes you wish to read and write, i.e., read `nitems` each 1 byte in size.

- **File Seeking**: Seeking in a file is the primary reason to use a file pointer. This is a procedure that allows you to move the read head to arbitrary locations in the file. Here is the core function:

```
int fseek(FILE * stream, long offset, int whence);
```

  which moves the read head of the file stream to the `offset` (measure in bytes) from a starting postion `whence`. For example, to seek 100 bytes into a file, use this seek call:

```
int fseek(FILE * stream, 100, SEEK_SET);
```

  Where `SEEK_SET` refers to the start of the file. You may find the following functions also useful: `ftell()`, which reaturns the current file position; and `rewind()`, which resets the file read position to the start of the file.

- **Converting to File Descriptor**: Finally, I should note that you can always convert a file pointer to a file descriptor and vice versa using either `fdopen()` and `fileno()`.

Socket Programming API

Although you've been using sockets in previous labs, you will need additional functionality to complete this lab. Note, that you will be using standard stream sockets, TCP connections, and not raw sockets.

- **Opening a socket for streaming:** To open a stream socket, you will use the following function call.

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

  A call to `socket()` takes a socket domain, e.g., internet socket `AF_INET`, the type of the socket, `SOCK_STREAM` indicate a stream session or TCP, and a protocol, which is 0 for `SOCK_STREAM` over IP. `socket()` returns a socket file descriptor, which is just an integer.

- **Connecting a socket:** In previous labs, you were using connection-less sockets where you just send data to a particularly destination without establishing a connection *a priori*. With stream sockets you must first connect with the destination before you can begin transmitting data. To connect a socket, use the `connect()` system call:

```
int connect(int socket, cost struct sockaddr *address, socklen_t address_len)
```

  which takes a socket file descriptor, a pointer to a socket address, and the length of that address. The return value of `connect()` indicates a succesful or failed connection: You should refer to the manual for more details on error conditions.

  Regarding socket addressing: you'll probably want to use the `sockaddr_in` form of `sockaddr`, which is the same size and has the following structure members:

```
struct sockaddr_in {
        u_char  sin_len;
        u_char  sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

And an in_addr is a uint32, or a 32 bit number, like an int. You can convert string representation of internet addresses to in_addr's using either getaddrinfo() or gethostbyname(). Sample code is available on the message board.

Note that if your computer has mutltiple interface, you must first bind() your socket to one of the interfaces using the bind() system call. It has the following function definition:

```
int bind(int socket, struct sockaddr *address, socklen_t address_len)
```

where the address indicates which of the servers interfaces, identified by IP address, this socket should use for listening (or sending).

- **Listening and Accepting a Connection:** On the server end, a socket must be set such that it can accept incoming connections. This occurs in two parts, first it requires a call to listen(), and second, a call to accept(). The function definition for listen() is as follows:

  ```
  int listen(int socket, int backlog)
  ```

  Of course, listen() first argument is the socket that the will be listened on. The second argument, backlog indicates the number of *queued* or *backlogged* incoming connections that can be pending waiting on an accept() call before a connection refused message is sent to the connecting client.

  The accept() function is the key server side mechanism of socket programming. Let's start by inspecting its function definition:

  ```
  int accept(int socket, struct sockaddr * address, socklent_t * address_len)
  ```

  Essentially, given a socket that is listening to incoming connections, accept **will block** until a client connects, filling in the address of the client in address and the length of the address in address_len.

  The return value of accept is very important: It returns a *new socket file descriptor* for the newly accepted connection. The information about this socket is encoded in address, and all further communication with this client occurs over the new socket file descriptor. Don't forget to close the socket when done communicating with the client.

- **Reading and Writing:** Reading and writing from an stream socket occurs very much like reading and writing from any standard file descriptor. There are a number of functions to choose from, I suggest that you use the standard read() and write() system calls. Writing is rather simple:

  ```
  ssize_t write(int filedes, void * buf, size_t nbyte)
  ```

  which, given a file descriptor (the socket) and a buffer buf, write() will write nbyte's to the destination described in the file descriptor and return the number of bytes written.

  Reading from a socket is slightly more complicated because you cannot be certain how much data is going to be sent ahead of time. First consider the function deffition of read():

  ```
  ssize_t read(int filedes, void * buf, size_t nbyte)
  ```

  Similar to write(), it will read from the give file descriptor (the socket), and place up to nbyte into the buffer pointed to by buf, returing the number of bytes read. However, consider the case where the remote side of the socket has written more than the buffer size of bytes. In such cases, you must place the read() in a loop to clear the line. Here is some sample code that does that:

```
while(read(sockfd, buf, BUF_SIZE)){

 //do something with data read so far

}
```

That is, the loop will continue until the amount read is zero, which indicates that there is no more data to read. But be careful, subsequent calls to `read()` when there is no data on the line will block until there is something to read.

Finally, a note about the manual pages for `read()` and `write()`, the relevant manuals are in section 2 of the manual, which you access this way:

```
#> man 2 read
#> man 2 write
```

- **Closing a Socket:** To close a socket, you simply use the standard file descriptor `close()` function:

```
int close(int filedes)
```

Putting it all together

Given all of these functions above, it might be hard for you to see how everything works together. Below, I've provided some psuedocode for a server and a client to help that intuition:

Client:

```
int sockfd;
struct socckaddr_in sockaddr;
char data[BUF_SIZE];

//open the socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);

//set socket address and connect
if( connect(sockfd, &sockaddr, sizeof(sturct ockaddr_in) < 0){
  perror("connect");
  exit(1);
}

//send data
write(sockfd, data, BUF_SIZE);

//close the socket
close(sockfd);
```

Server:

```
int serve_sock, client_sock, client_addr_size;
struct sockaddr_in serv_addr, client_addr;
char data[BUF_SIZE];

//open the socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);

//optionally bind() the sock
bind(sockfd, serv_addr, sizeof(struct sockaddr_in));

//set listen to up to 5 queued connections
listen(sockfd, 5);

//could put the accept procedure in a loop to handle multiple clients

//accept a client connection
client_sock = accept(sockfd, &client_addr, &cleint_addr_len);

while(read(client_sock, data, BUF_SIZE)){
  // Do something with data
}

//close the connection
close(client_sock);

}
```