

# CS 452 - Lab 1: Socket Programming: **fsnd**

Due: 04-Oct-2016

## Overview

The objective of this assignment is to gain experience in application level programming with stream (TCP) sockets with a client/server socket application named `fsnd` (file send). You will also learn the usage of advanced file manipulation in C using file pointers and file streams.

You should develop your code in C and on any CS lab machine or your personal machine, but all code must compile and run on machines in the P107 lab.

## Deliverables

Your submission should minimally include the following programs and files in a folder named `<emailname>_lab1`:

- `fsnd`
- `Makefile`
- `README`

The `README` file must include a short header containing your name, username, and the assignment title. It should also include a short description of your code, tasks accomplished, and how to compile, execute, and interpret the output of your programs. Any questions posed in this lab write-up should contain answers your `README` file.

## **fsnd** (file send utility)

In this lab you will reacquaint yourself with socket programming and C file I/O primitives by implementing a form of `fsnd` for file transferring. First, a description of `fsnd` is provided followed by a description of the lab requirements for implementing `fsnd`.

## Basics of **fsnd**

Your `fsnd` program is a simple networking program that connects to a remote server and sends input from `stdin` or a file input 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 an output file or `stdout`. `fsnd` can be viewed as a data transport over-the-network utility. Here is an example of some of its usages:

- **fsnd as a client:**

The most common usage of `fsnd` is as a client; that is, as a program that connects to a remote server. It sends the file as a stream to the server and the server sends them to `stdout`. The `fsnd` utility can be used in two modes. You can use `fsnd` by specifying the name of a file, e.g.:

```
$ fsnd thing1.cs.uwec.edu 9285 testfile
```

or by piping a stream of data to it, e.g.:

```
$ echo "Hello World" | fsnd thing1.cs.uwec.edu 9285
```

Note that the syntax of `fsnd` on the client side is different from the one on the server side as shown below. This above command will connect to `thing1.cs.uwec.edu` on port 9285.

Here is another example of using `fsnd` as a client to read input in the form of a file. For example:

```
$ fsnd -p 9285 thing1.cs.uwec.edu alpha.txt
```

The `fsnd` command is issued such that it connects to `thing1` on port 9285 and uses the file `alpha.txt` as the data to send.

In addition to this standard usage, `fsnd` has the additional feature of being able to send parts of a file, the part of `fsnd`. Consider a large file, such as the text of *Great Gatsby*, and you only want to send the first 512 bytes of the file to a remote server rather than the whole file. You can use this `fsnd` command:

```
$ fsnd -n 512 thing1.cs.uwec.edu great_gatsby.txt
```

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

```
$ fsnd -o100 -n 512 thing2.cs.uwec.edu great_gatsby.txt
```

Which will send 512 bytes of the file starting at byte 100. That is, it will send bytes 100 through 611 of the file.

## **fsnd as a server:**

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

```
$ fsnd -l -p 9825 <output file>
```

Here, `fsnd` is provided two command line arguments (a.k.a. options or flags): `-p`, which indicates what port to open the socket on; and `-l`, which indicates that `fsnd` should listen for incoming connections. You can test your `fsnd` server by trying to connect to it using another instance of `fsnd`. **By default, output is sent to stdout unless and output file is listed in the command.**

If you issue the following command on the client side to the server (`thing1.cs.uwec.edu`), you type:

```
$ echo "Hello World" | fsnd thing1.cs.uwec.edu 9285
```

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

Note once again that by default, the server will dump everything it receives to `stdout`. If you want the output to be written to a file you must specify the output file name and the server will redirect the received stream to the appropriate destination and does not care about the offset or the number of bytes it receives when writing. It will just open the output file for writing, clobbering whatever data that already exists.

You and your lab partner, will implement a form of `fsnd` for transferring partial or whole files. Here are flags that must be implemented for the `fsnd` utility:

```
fsnd [OPTIONS] dest_ip file
    -h                Print this help screen
    -v                Verbose output
    -p port            Set the port to connect on (e.g., 9285)
    -n bytes           Number of bytes to send, defaults whole file
    -o offset          Offset into file to start sending
    -l                Listen (on server side) on port instead of connecting and
                     write output to file and dest_ip refers to which ip to bind to.
                     (default: localhost)
```

**EXTRA CREDIT 10 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 `fsnd` will write the 512 bytes to the output file at the appropriate offset, and, on successive runs of `fsnd` for the same output file, it will not clobber the file but continue to write data to the appropriate location in the file. Add an options flag, `-m` for “maintain”, that will enable this functionality.

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

## Required format of a session between client and server

Assuming that we connect from one of our VMs to another, transferring a file called `arya.txt`. The file size is 1G:

Client (when verbose flag is set)	Server
\$ <code>fsnd -v -p 9825 -o 100 -n 512 thing2.cs.uwec.edu arya.txt</code>	\$ <code>fsnd -l -p 9825</code>
Connecting to IP: 10.35.195.47 On port: 9285 File name: arya.txt File Size: 1048576000 Sending file [offset=100   send size = 512] Percentage sent: [10...15...20...25...30.....] Send Success!	Listening on port: 9825  Receiving file: arya.txt Percentage rec'd: [10...15...20...25...30.....] Receive: success File size received: 512
MD5: (Only for entire files and not partial) 7d41700dbc59cd95c3076a47eb168e6c	MD5: 7d41700dbc59cd95c3076a47eb168e6c

The MD5 hash is a digital signature utility that produces a 128-bit unique number that is used to authenticate the integrity of the files you have transferred over, i.e., the file at the source must have the

same md5 hash value as the one at the destination.

To use the md5 hash on our linux machines:

```
$ md5sum arya.txt
7d41700dbc59cd95c3076a47eb168e6c arya.txt
```

## File Programming Preliminaries and Socket Programming

Below are some brief descriptions of the requisite C functions needed for this lab. First, file I/O is discussed, particularly file streams, and following, the basics of socket programming is discussed, including C-style pseudo-code examples.

### File Streams

- **Opening a file:** You are probably already familiar with the basic file opening procedure `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 char * filename, const char * 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.

- **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 position `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 returns 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

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);
```

`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 particular 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, const 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 successful 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_in`, 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 multiple 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 server's 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, socklen_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 definition 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**, returning 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.

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

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

## Putting it all together

Below are examples in skeletal psuedocode for a server and a client :

### Client:

```
int sockfd;
struct sockaddr_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(struct sockaddr_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

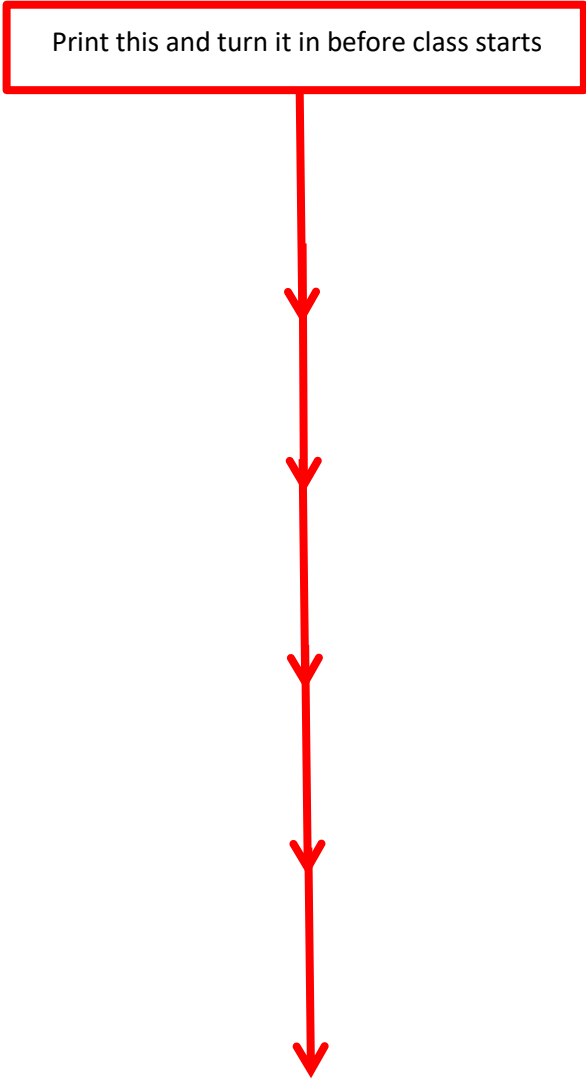
//accept a client connection
client_sock = accept(sockfd, &client_addr, &client_addr_len);

while(read(client_sock, data, BUF_SIZE)){
// Do something with data
//close the connection
close(client_sock);
}
```

## Lab Submission

1. Zip or tar folder and put it in the W drive. Name your folder: <emailname>\_lab1. Include a makefile and README in the folder.
2. Turn in a hardcopy of the banner page (see below). Do not turn in code.
3. Email me the percentage workload of every person in the team. By default, if no email is received, workload is assumed to be zero, even if your partner says otherwise.
4. Schedule a demo in P107.

Print this and turn it in before class starts





# CS 452 – OS Lab Assignment Cover Page

Name(s) \_\_\_\_\_ Lab#: 1

1. Include a copy of your source code, a design document (if required), and a sample run
2. Fill in the table of contents including the names of routines and the corresponding pages.
3. Indicate the status of your program by checking one of the boxes.
4. Submit the assignment at the beginning of the class on the due date.

**Program Status:** (check one box)

- |   |  |
|---|--|
| <input type="checkbox"/> Programs run with user-defined test cases. | <input type="checkbox"/> Program run with some errors. |
| <input type="checkbox"/> Programs compile and run with no output.   | <input type="checkbox"/> Programs do not compile.      |

Grading:	Points	Score
Program correctness/Design	65	
Stress Test	35	
Extra Credit	<10>	
Total	100	

## Honor Code Department of Computer Science University of Wisconsin – Eau Claire

As members of the University of Wisconsin – Eau Claire community and the computer science discipline, we commit ourselves to act honestly, responsibly, and above all, with honor and integrity in all areas of campus life. We are accountable for all that we say, read, write, and do. We are responsible for the academic integrity of our work. We pledge that we will not misrepresent our work nor give or receive illicit aid. We commit ourselves to behave in a manner which demonstrates concern for the personal dignity, rights and freedoms of all members of the community and those that depend on the expertise we possess.

For all course work in the Department of Computer Science, students will write and sign (if printed) the following: **“I have abided by the Department of Computer Science Honor Code in this work.”**

I accept responsibility to maintain the Honor Code at all times.

Names      1. \_\_\_\_\_ 2. \_\_\_\_\_

Signature      1. \_\_\_\_\_ 2. \_\_\_\_\_

Date      \_\_\_\_\_ / \_\_\_\_\_ / 2016 \_\_\_\_\_