ECE361 File Transfer Lab Tutorial

Getting Started

Organizing Your Development

The file transfer lab is broken down into 3 sections, each due in a different practical. This lab becomes much more manageable once you break it down into small pieces that you can test along the way! To get started, we recommend the following:

1. Create a `deliver.c` and `server.c` file, each with a main function that prints “hello world”.
2. Create a makefile that can build them, then run each program to make sure it is working.
3. Incrementally add the new functionality (e.g., process the runtime arguments, check for existence of the file, figure out how to create sockets, etc.), and remember to test as you go.

Keep in mind that your code must be able to run on the UG machines; as such, we encourage remote development using a VNC session, SSH on a terminal, or SSH through an IDE (e.g., VS code) in case physical access is unavailable. If you ever get stuck, remember that the teaching team is always happy to help!

```
CC=gcc
all: server deliver
server: server.o
deliver: deliver.o
clean:
	rm -f *.o server deliver
```

Figure 1. Sample makefile

We highly recommend that you use a version control system such as GitHub to back up your work. The GitHub student developer pack is free and gives you access to private repositories. GitHub can also help with dividing tasks among group members through its project management features. If case you’ve never used it before, here are some useful links:

- Student developer pack: https://education.github.com/pack
- Getting started with GitHub: https://guides.github.com/activities/hello-world/

Testing

You are required to run the client and server on separate UG machines when testing your code. You can do this remotely by opening two different terminals, each of which is logged into a different UG machine. As a tip, suppose you are logged into ug140; then you can easily “switch” to ug141 by typing `ssh ug141` in the terminal.

Now that you are logged into two different UG machines, you need to start your server on one of them and your client on the other. Let’s say the server machine is ug140 and the client machine is ug141. When starting the client, you need to input the address of the UG machine the server is running on, as well as the port. There are 2 options for how to enter the server’s address:

1. Input the server’s IP address in dotted-decimal form. To get the server’s IP address, run the command `hostname -I` in the server machine’s terminal. Following our example, we enter this command in the ug140 terminal and see that the IP address is 128.100.13.140.
    a. In your client code, this IP address will be passed into main as a string. Since it’s a string, it’s said to be in “printable” form. We can use the function `inet_pton` to convert this

string from **p**rintable form **to** "**n**etwork" form (32-bit unsigned long). The network form is what's needed when working with sockets.

2. Input the server's address in human-readable form, e.g., ug140 or ug140.eecg.utoronto.ca.
    a. In your client code, you will need to perform a DNS lookup to convert this human-readable string into an IP address. Refer to the section on `getaddrinfo` for details.

You can choose to support one or both of the above options.

Once your code is running, you may find it doesn't behave the way you expect. While there are various ways to debug code, inevitably many students end up using print statements. As a quick tip, it might be helpful to record your program output in a log file, e.g., `./server 55000 2>&1 >log.txt` will run the server on port 55000, and redirect both `stdout` and `stderr` to a file called log.txt.

Readings

The networking libraries can be daunting at first, but the essential functions to focus on are: `socket`, `bind`, `sendto`, and `recvfrom`. **These are explained in the** Function Reference **section below**.

Additionally, if you ever want information about a C function, you can use the `man` command in the terminal (assuming you are using a Linux machine). For example, `man socket` will present you with the Linux manual page for `socket`.

As for Beej's guide, some of the sections like "What is a socket?", "IP Addresses, structs, and Data Munging" are good for understanding the background required to use the networking libraries. "System Calls or Bust" is good to skim through, then refer to when needed. The "Datagram Sockets" section under "Client-Server Background" is the most relevant sample code for this lab since **you're required to use UDP**. We recommend reading a bit of Beej's guide to get an idea of how your code should work, but don't spend too long on it—trying to understand everything before starting to code may not be a good use of your time, since things usually sink in better through your own trial and error.

When you're ready, dive into writing code for the lab, referring to relevant sections of this document and Beej's guide as needed, and searching Google or Stack Overflow for specific problems that arise (or posting on Piazza!). There is also an FAQ coming up which answers the most common questions!

Some C Programming Advice

During the previous offerings of this course, we observed some common errors students make when programming in C. We offer the following as reminders/advice:

- When you declare a variable, make sure you **initialize it before using it**. Using uninitialized variables can lead to unexpected behaviour that is painful to debug. Structs can be initialized using memset or array initialization syntax, i.e., `struct sockaddr my_addr = {0};`.
- *Declaring* a pointer does **NOT** cause memory to get allocated. For example, if you want to create an array and write some values into it, you either need to declare a fixed-length array somewhere in your code, or use malloc to get the OS to allocate the memory for the array at runtime.

FAQ

General/Section 1

Q: Are we allowed to use code from Beej's guide?
A: Yes. Make sure you cite your source in somewhere in your comments!

Q: Is UDP connectionless? If so, does that mean we do not need to form a connection (i.e., do not need to use connect(), listen() and accept()) in order to send messages from client to server?
A: Correct!

Q: `recvfrom` gives me a bad address error or is complaining about an `EFAULT` when I check `errno`. What could be wrong?
A: Double-check that you are passing the correct arguments into `recvfrom`. Are the things that are supposed to be pointers actually pointers? Similarly for values. Is the amount of data you are trying to read larger than the buffer you are storing the data in?

Q: How should I prepare for lab questions from the TA?
A: Understand your code, even the parts your partner worked on, and think about how your code connects to concepts from lecture. If you don't fully understand what the TA is asking you, ask them to clarify!

Q: Do the files that we transfer need to be in the directory where the deliver program is running?
A: That's entirely up to you. You can also choose whether your program works with relative file paths or absolute file paths.

Section 2

Q: Is there a specific file we need to transfer?
A: No, but you must be able to demonstrate the transfer of binary files such as images or PDFs. For example, demonstrating that you can transfer the lab handout would be perfectly fine.

Q: My code works for non-binary data but breaks for binary data! What could be wrong?"
A: The most likely issue is that you're using a string function (e.g., `strlen`, `strcpy`, `fgets`, `fputs`, `sprintf`, etc.) somewhere that you shouldn't be.

Q: Do we need to convert messages into packet structs when receiving (i.e., at the server)? Or can we work with the received array directly?
A: We strongly recommend you convert received messages into packet structs. This is a good practise, and if you choose not to do it then make sure you have justification.

Q: Can we store file fragments in a fixed size char array instead of dynamically allocating fragments everytime we need to send them?
A: Absolutely. You can choose to use malloc for this lab, but there's no need to. If you *do* choose to use malloc, make sure you're not reading the entire file onto the heap all at once—for large files (e.g., 500 MB) this would be a very inefficient use of your memory.

Section 3

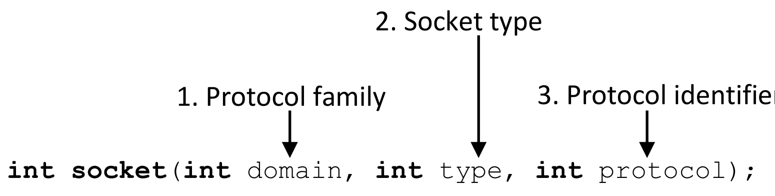
Q: `recvfrom` complains about `EAGAIN`/`EWOULDBLOCK`. What could be wrong?
A: This can happen when you are using non-blocking sockets. In this case, this error is not fatal, it's just an additional way of informing your application that no data has been received yet.

Function Reference

socket

A socket is your gateway to and from the internet. In this lab, we focus on UDP sockets, which are message-oriented. This means that one call to `sendto` at the transmitter corresponds to one call to `recvfrom` at the receiver. In later labs, we will see that this is not the case with TCP.

To start communicating with others as a client, you first create a socket. If you want to send a message, then you give it to the socket and tell it the destination; the socket library will take care of the details. Once you've created a UDP socket, it will always be listening for messages in the background while you go about other things in your code, and when messages are received, they will be added to a queue. So, when you're programming, receiving a message from a UDP socket just comes down to checking whether there's anything in the message queue.



```
int socket(int domain, int type, int protocol);
```

Arguments

1. Specifies the protocol family used for communication. In our labs, we want to use the IPv4 internet protocols, so pass in `AF_INET`.
2. This argument controls whether the socket is UDP, TCP, or something else. For this lab we **must** use UDP, so pass in `SOCK_DGRAM`.
3. Used to choose among different protocols that the protocol family supports for the given socket type. Since there is only one UDP protocol within the `AF_INET` family, just pass in 0.

Return Value

Returns a file descriptor (FD) for the new socket. You will pass this FD into all other socket functions.

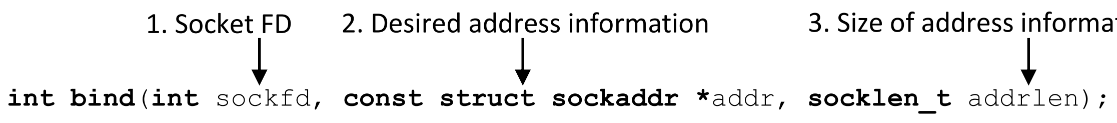
References

- https://man7.org/linux/man-pages/man2/socket.2.html
- https://man7.org/linux/man-pages/man7/ip.7.html

bind

`bind` associates a socket with an address (IP address + port number). Usually, a client program does not care about its socket address, but this is not the case for a server program. The server's socket can't just be associated with some random address, otherwise clients won't know how to reach it.

To solve this problem, the server uses bind to set its socket address in a custom fashion. Often, we are okay with using any IP address available to us, and what we care about is specifying the port number.


1. Socket FD
2. Desired address information
3. Size of address information

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

Arguments

1. The socket FD tells `bind` which socket you want to try to bind to an address.
2. **(POINTER)** Address information, explained in the section on `sendto`, i.e., Figure 2. Usually, we set `sin_addr = htonl(INADDR_ANY)` to bind the socket to all local interfaces, and we set `sin_port` to the port number we want to try to bind to (remember to use `htons`).
3. Size of the `struct sockaddr` passed in for argument 2.

Return Value

Returns 0 on success and -1 on failure. **You should always check the return value of `bind`.** Even though there are tens of thousands of port numbers to choose from, you will be surprised at how often some other application on your machine is using the same port number as you. If `bind` fails, just try again with a different port number, or let the OS choose an available port for you automatically.

References

- https://man7.org/linux/man-pages/man2/bind.2.html
- https://man7.org/linux/man-pages/man7/ip.7.html

getaddrinfo

`getaddrinfo` is used when you know the (human-readable) hostname of the machine you're trying to reach (e.g. "ug140"), but you don't know the machine's IP address. `getaddrinfo` is useful in this case because it performs a DNS query to translate the name into an IP address. This query can return multiple results in the form of a linked list, because a hostname can map to multiple addresses (e.g., IPv4 and IPv6).

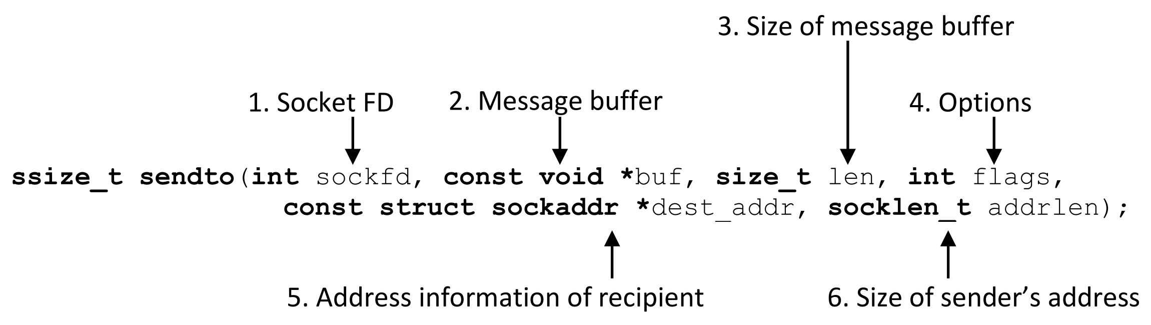
References

- https://man7.org/linux/man-pages/man3/getaddrinfo.3.html
- Section 5.1 in Beej's Guide: https://beej.us/guide/bgnet/html/#getaddrinfoprepare-to-launch

sendto

`sendto` is used to make a socket send a message. If the socket is not already bound to an address when `sendto` is called, then it will automatically be bound to an available address.

When programming, make sure you are careful about which arguments are pointers and which are values!

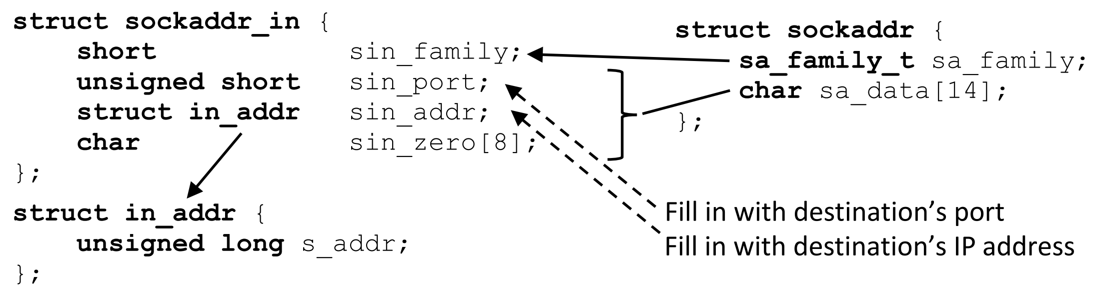

3. Size of message buffer
1. Socket FD
2. Message buffer
4. Options
5. Address information of recipient
6. Size of sender's address

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                  const struct sockaddr *dest_addr, socklen_t addrlen);
```

Arguments

1. The socket FD tells `sendto` which socket you want to use to send messages.
2. **(POINTER)** Pointer to the message you want to send (usually a `char` array).
3. Length of the message you want to send (i.e., number of bytes).

4. This argument allows you to specify transmit options. For this lab, you can pass in 0 (we don't need any options).
5. **(POINTER)** You need to provide `sendto` with the address information of the recipient, so that it knows where to deliver the message. First, create a `struct sockaddr_in` on the stack (the "in" stands for internet, since we are using that protocol suite), and set `sin_family = AF_INET` ("internet address family"). Next, you need to set `sin_port` and `sin_addr` to the value of the port and IP address of the recipient, respectively. There's a catch though: you must convert these values from the *host byte order* to the *network byte order*, using functions such as `htons` and `htonl` ("host to network short/long"). Once all that is done, pass the struct into `sendto` via pointer (you will have to typecast it to `struct sockaddr*` when you do this).



```
struct sockaddr_in {
    short              sin_family;
    unsigned short     sin_port;
    struct in_addr     sin_addr;
    char               sin_zero[8];
};

struct in_addr {
    unsigned long s_addr;
};
```

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[14];
};
```

Fill in with destination's port
Fill in with destination's IP address

Figure 2. Address information struct

6. Size of the `struct sockaddr` passed in for argument 5.

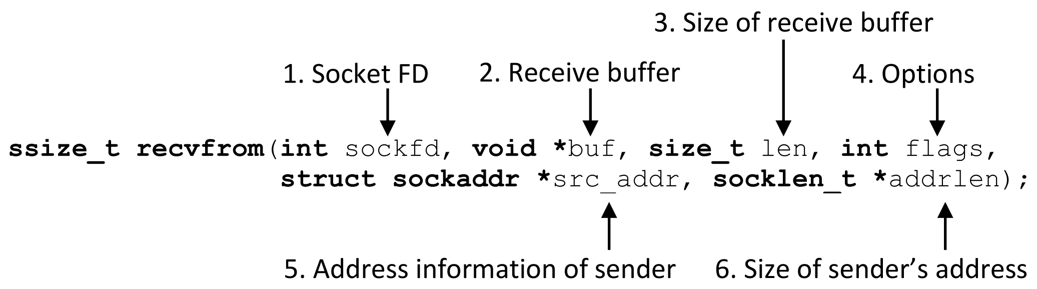
Return Value
Returns the number of bytes sent.

References
- https://linux.die.net/man/2/sendto
- https://linux.die.net/man/3/htons
- https://www.gta.ufrj.br/ensino/eel878/sockets/sockaddr_inman.html

recvfrom
`recvfrom` is used to receive messages from a socket. It is one of the functions students struggle the most to use properly. You should think of it as a way to (1) read a single message from the UDP message queue, and (2) get information about who sent the message so that you can reply to them.

`recvfrom` is a blocking call by default, meaning that it does not return until at least one message has been received. On the other hand, suppose that 10 messages have already been received before we call `recvfrom`. Then when we call `recvfrom`, it will read the first message from the queue into the receive buffer and return immediately. In this case, 9 messages will remain in the message queue and can be retrieved by calling `recvfrom` more times.

When programming, make sure you are careful about which arguments are pointers and which are values!


3. Size of receive buffer
1. Socket FD
2. Receive buffer
4. Options
5. Address information of sender
6. Size of sender's address

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

Arguments (inputs)

1. The socket FD tells `recvfrom` which socket you want to receive messages from.
2. **(POINTER)** You must provide `recvfrom` with a buffer for it to copy received messages into. This receive buffer can be an array on the stack, or you can use malloc to create an array on the heap. In the former case, you can determine the length of the array using `sizeof(buf)`, while in the latter case you need to keep track of the array length manually (you will need this length soon).
3. `len` is ***not*** the number of bytes you "wish to receive" (that is the wrong way of thinking!) Rather, `len` is the length of your receive buffer. At most, `recvfrom` will copy `len` bytes into `buf` (the exact number of bytes copied depends on the length of the message received). If the size of the received message exceeds `len`, the excess bytes will be discarded (UDP is not known for its reliability, after all…).
4. This argument allows you to specify some options, e.g., enabling non-blocking receptions. For this lab you can pass in 0 (we don't need any options).

Arguments (outputs)[1]

5. **(POINTER)** You need to create an empty `struct sockaddr_storage`[2] on the stack and pass it into `recvfrom` via pointer (you will have to typecast it to `struct sockaddr*` when you do this). Doing so will result in `recvfrom` automatically populating it with the address information of the sender (see Figure 2). You can reply to the sender by passing this populated struct into `sendto`.
6. **(POINTER)** This is both an input and an output. You need to create a `socklen_t` on the stack, initialize it to the size of the `struct sockaddr_storage` you created for argument 5, and pass it into `recvfrom` via pointer. `recvfrom` will modify it to match the length of the sender's address information.

Return Value

Returns the number of bytes received.

References

- https://linux.die.net/man/2/recvfrom

---

[1] You might be wondering why these arguments are called "outputs". Basically, `recvfrom` would like to have more than one return value, but it can't due to limitations of the C programming language (if you use sockets in Python, you will see that `recvfrom` has multiple return values). In C, the way to get around this is to pass in the additional return values by pointer and modify them within the function.

[2] A `struct sockaddr_storage` is a container that's big enough to hold all possible types of addresses (e.g., IPv4 as well as IPv6). For this lab, you can think of it as a `struct sockaddr_in`.