File Transfer Lab Exercises

ECE361

Student ID: 1005689480 Date: 2022-02-03

Due: 11:59 PM, February 3rd, 2022

Instructions

This assignment is out of 20 marks, for a total of 2.5% of your final grade. It's not intended to be hard; it's intended to get you thinking about the labs (tip: you may find the File Transfer Tutorial PDF useful). Solutions will be posted to Quercus on February 4th. If (and only if) you receive a passing grade on this assignment, then you will have 1 week from the time your grade is posted to submit corrections, if desired. Corrections should also explain why your previous answer was incorrect.

Exercises

Socket Programming (10 marks)

1. Why does a server need to `bind`? Does it do this before or after it starts communicating with clients? (**2 marks**)

Bind associates a socket with an address (IP address + port number).
The server need to do this in order to hear from a client. The client is sending to this port number on this IP address.
It must do this before it start communicating with the clients

2. Does a client program also need to `bind`? If so, when does it do this? If not, why not? (**2 marks**)

The client program do not need to bind.
Because we are using unconnected DATAGRAM. Only the server need to hear from the client. The client do not need to hear from the server.

3. A common misconception is that you need to call `recvfrom` before the sender transmits their message, otherwise you won't receive anything. Please explain why this is not the case. (**1 mark**)

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. When messages are received, they will be added to a queue.
So, when you're programming, you can use recvfrom() to check whether there's anything in the message queue. It doesn't have to be called before the sender transmit their message. If recvfrom() isn't called before the sender send message, the message will be queued up for the socket.

4. Fill in the missing arguments to `recvfrom` below. (**2 marks; 0.5 each**)

```
char buf[1024];
struct sockaddr_storage their_addr;
socklen_t addr_size = sizeof(their_addr);
recvfrom(sockfd, A1, A2, 0, A3, A4);
```

A1 = buffer to copy received messages into

A2 = length of the receive buffer

A3 = pointer to a local struct sockaddr_storage or addrinfo, passed by reference

A4 = pointer to a local int that should be initialized to sizeof(struct sockaddr_storage or addrinfo)

5. The code below contains 3 issues or bad practises (that we know of). Identify them and explain. (**3 marks**)

```
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_port = 54321;
server.sin_addr = "128.100.13.140";
socklen_t addr_size = sizeof(server);
char msg[] = {'h', 'e', 'l', 'l', 'o', '\0', 'h', 'i'};
sendto(sockfd, msg, strlen(msg), 0, (struct sockaddr*)&server, addr_size);
```

sin_port should use htons(), not directly assigned to an int

sin_addr should be a structure in_addr, not a string

char msg[] = {'h', 'e', 'l', 'l', 'o', '\0', 'h', 'i'};
Null character mark the end of the string, thus strlen(msg) would only return the length of "hello", which is wrong. Should place the null character to the end of the character array.
corrected: char msg[] = {'h', 'e', 'l', 'l', 'o', 'h', 'i', '\0'};

Serialization and Deserialization (10 marks)

Serialization refers to the process of converting a struct into a sequence of bytes in preparation for transmission. Deserialization is performed at the receiver and refers to the reverse process.

In general, it is not a good idea to pass structs directly into `sendto`. In the following exercises, we will ask you to think about why. Hint 1: why do we use functions such as `htons` and `htonl`? Hint 2: what issues might we run into when trying to send a pointer to another machine or process?

6. For each of the following structs, indicate whether it is okay to pass it directly into `sendto` and provide a brief justification. (**2 marks each; 0.5 for correct YES/NO answer and 1.5 for justification**)

`struct Node1 {`
`    uint16_t byte;`
`};`
	YES / [✓ NO]	Different machines have different byte order, usually little Endian or big Endian. We should convert to the network byte order using htons() before sending.

`struct Node2 {`
`    int age;`
`    char name[64];`
`};`
	YES / [✓ NO]	Same issue as above.
		Both int and char have different bytes order and also different size in different machines.

`struct Node3 {`
`    int age;`
`    char* name;`
`};`
	YES / [✓ NO]	Same issues as above
		Also, we can't pass pointers in send because pointers point to address in the host. When the message arrive at the receiver, pointers has no meaning.

`struct Node4 {`
`    int data;`
`    struct Node4* next;`
`};`
	YES / [✓ NO]	Same issue as above.

7. Is a Node3 harder to serialize than a Node4? Why or why not? (**2 marks**)

Node 4 is harder to serialize than node 3.
Both nodes are structs and contains pointer. However, node3's size could be easily determined, by looking at the char* name. Node4's size is hard to determine because we have to traverse through the linked list to find its size. We need the size in order to convert a struct into a sequence of bytes in preparation for transmission. Thus, node 4 is harder.