File Transfer Lab Exercises—Solutions

ECE361

Exercises

Socket Programming

1. Why does a server need to `bind`? Does it do this before or after it starts communicating with clients?
    a. Answer: The main duty of a server is to respond to requests from clients. This means that clients must know how to reach the socket, i.e., the socket must somehow be associated with an address that is mutually agreed upon between the clients and server. For this reason, a server first creates a socket, then uses `bind` to associate it with an address *before* engaging in any communications.
2. Does a client program also need to `bind`? If so, when does it do this? If not, why not?
    a. Answer: A client program *does* need to `bind` —if it didn't, then programs it talks to would have no way of replying to it, since it wouldn't have an address! The subtlety is that you often don't see manual calls to `bind` in a client program, because clients don't usually have preferences among addresses. Instead, what happens is a client will call a function such as `sendto`, and the socket library will automatically `bind` the socket to an available address under the hood. This address will persist until the socket is closed.
    b. Note: 1 mark was given for answers that explained why it's usually not necessarily to call `bind` in a client application. However, for full marks, answers had to also identify the fact that binding *does have to occur* at some point prior to communication, even if it's not a manual step (i.e., just because you don't have to call `bind` in the client application doesn't mean binding doesn't occur under the hood).
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.
    a. Answer: As mentioned in the socket section from the tutorial document, a UDP socket continuously listens for messages in the background and maintains a message queue internally. So, the socket will receive messages even before `recvfrom` is called—`recvfrom` just reads a message if there is one, and if not then it waits.
4. Fill in the missing arguments to `recvfrom` below.

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

    Answer:

    A1 = `buf`

    A2 = `1024 or sizeof(buf)`

    A3 = `(struct sockaddr*)&their_addr`

    A4 = `&addr_size`
5. The code below contains 3 issues or bad practises (that we know of). Identify them and explain.

```
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);
```

Answer:

1. Missing network byte order conversion when assigning `sin_port` a value. Should use `htons(54321)`.
2. `sin_addr` should not be assigned a string! When we have the IP address as a string, we first need to use a function such as `inet_pton` to convert it to a 32-bit value.
3. `strlen`(msg) will return 5 while `sizeof`(msg) will evaluate to 8 (includes all characters). It is thus an issue to use `strlen`! In this course, you should always assume that buffers contain binary data, so be careful about string functions—they could cause you to lose data!

Other acceptable answers:

- `memset` has not been used on the `server` struct to initialize its contents to 0; could lead to issues due to uninitialized data
- `server.sin_addr` is itself a struct; should use `server.sin_addr.s_addr` instead

Serialization and Deserialization

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.

`struct Node1 { uint16_t byte; };`	YES / **NO**	Justification: different architectures can interpret multi-byte data types differently (endianness), so we should convert the `byte` member to network byte order before sending. Struct padding (to force alignment) can also vary across compilers.
`struct Node2 { int age; char name[64]; };`	YES / **NO**	Justification: same as above. Additionally, an int is not guaranteed to be the same length across platforms. Furthermore, sending all 64 chars is inefficient if some of them are unused. Note: we don't need to worry about endianness for a string because even though strings are multiple bytes, the individual characters are only a single byte each. We

		only need to worry about endianness for primitive types, and char is a special case.
`struct Node3 { int age; char* name; };`	YES / **NO**	Justification: same as above. Additionally, it doesn't make sense to send a pointer across the network because pointers refer to local memory and will generally differ in value (and size) across machines. Instead, the data being referenced needs to be communicated somehow.
`struct Node4 { int data; struct Node4* next; };`	YES / **NO**	Justification: same as above.

7. Is a Node3 harder to serialize than a Node4? Why or why not?
    a. Answer: the way that you justify your answer is the most important part of this question. Regardless of whether you argued for Node3 or Node4 being harder, what aspects did you consider? What assumptions did you make? What was your metric for "hardness"? The best answers to this question presented serialization strategies for Node3 and Node4 and compared them.
    b. One might argue that since both structs consist of an int and a pointer they should be equally difficult to serialize. However, this relies on the fact that it is easy to serialize both int and char* (e.g., for `age`=22 and `name`="Alice", we could serialize Node3 by forming the string "`22:Alice`"), assuming that the length of the data pointed to by the char* is known. On the other hand, since Node4 is a linked list element, when we serialize the `next` member, we find that we are again tasked with the problem of serializing a Node4 (assuming `next` is not `NULL`)…and so on. For this reason, we argue that conceptually, it is harder to serialize a Node4.

       (So how do we serialize a Node4? One approach for serializing referenced data is called inlining or embedding. This is what we did above for the Node3 example, and this is also what's done in the example in section 2 of the file transfer lab handout. For the linked list, we could traverse it and form an array consisting of just the `data` members, send that, then reconstruct the linked list at the receiver by reading these values from the array in sequence and creating nodes as we go. Note that the values in the `next` field will be different at the transmitter and receiver, since these refer to local memory locations.)
    c. One could argue that it is easier to serialize Node4 since you can represent it as an integer array, whereas you need to combine two pieces of information when serializing Node3. Furthermore, if we don't know the length of the `name` field in Node3, then it might be impossible to serialize `name`.