UNIVERSITY OF DELAWARE
DEPARTMENT OF COMPUTER & INFORMATION SCIENCES

CISC 450 / CPEG 419: Computer Networks I

Fall Semester, 2018 Professor: Adarsh Sethi

Programming Project 1
Some Helpful Hints

Below are some helpful hints and suggestions for the Programming Project 1:

1. All programming for this course should be done on the host *cisc450.cis.udel.edu*. You can access this host if you have an *eecis* ACAD account. If you don't have such an account, fill out a request at *https://accounts.eecis.udel.edu*.

2. The client and server programs discussed in the class are available on the *cisc450* host in the directory */usa/sethi/networks/proj1*. You should copy the files in this directory to your own project directory and use these programs as building blocks for your own code.

3. You should change the port number SERV_TCP_PORT in the server program to a different port number chosen by you so you don't have a conflict with the port numbers being used by the other students in the class.

4. The socket API functions used in the TCP client and server programs (e.g., *send, recv, socket, connect*, etc.) have *man* pages that provide more detailed information on how to use these functions. It appears that these *man* pages are not loaded on our *cisc450* VM, but they are widely available on the Web. You can view them by typing *man send, man recv,* etc. in your browser search box.

5. You should know clearly the sizes of the different integer types on the *cisc450* VM, and use them appropriately. An *int* is the same as a *long int* and they are both 4 bytes long, while a *short int* is 2 bytes long.

6. The functions *htons*, *htonl*, *ntohs*, and *ntohs*, should be used on all short/long integers before putting them in the buffer for transmission, and after getting them out of the buffer when received. Strings and characters do not need such conversions. The conversion must be done individually on each integer; it cannot be done on a *struct* or other compound data structure.

7. The message to be transmitted by *send* does not necessarily have to be a string (or array of char). For example:

```
int i, j, bytes_sent;
scanf("%d", &i); // reads in a value for i
j = htonl(i);
bytes_sent = send(sock_client, &j, sizeof(j), 0);
```

The above will send out a 4-byte integer on the socket *sock client*. The same is true for the *recv* function as well. You can also directly send and receive a *struct*, in which case simply use the address of the *struct* as the second argument to the *send* and *recv* functions.

8. For this project, each message (request or response, as described in the project specification) should be transmitted as a single unit using only one invocation of the *send* command. It is not permissible to send each field of the packet in separate *send* commands. Similarly, use a single invocation of the *recv* command to receive a message.

9. Since the messages sent between the client and server may have many fields of different types, below are two suggestions on how to handle these messages so they are sent and received as a single unit (you may come up with other ways as well):

   - Declare a *struct* for your packet format that contains all the fields you want to have. Then you can directly use this structure in your *send* and *recv* function calls, similar to the code described in item 7 above. However, you have to be careful in placing the fields of the structure so they are aligned properly, e.g., an *int* should be aligned at 4-byte boundaries.
   - Use an array of characters as a buffer. Copy the message into this buffer when sending and copy it out of it when a message is received.

10. If you use the second method above, you must make sure that **each field is transmitted as a fixed number of bytes**. If you transform the field to a string to copy it into the buffer, it should be a fixed-length string and not a variable-length string. When converting an integer, the *wrong* method to do this conversion is to use *sscanf* and *sprintf*, because these functions transform a number into a variable-length string using its decimal representation. For example, if you say:

    ```
    int i;
    char sentence[256];
    i = 356780;
    sprintf(sentence, "%d", i);
    ```

    then the result will be that the string of characters “356780” consisting of 7 chars will be stored in *sentence* (the six digits in “356780” followed by the null char). The length of this string will vary depending on the value of *i*.

    The correct way of doing the conversion is to use the binary representation of *i*, which uses only 4 bytes, to produce a fixed-length (in this case, 4 bytes long) string. This can be done by using *memcpy* to copy a fixed number of bytes (in this case, 4) from *i* into a particular place within the sentence buffer. Don’t forget to cast both source and destination to *(void *)*.

11. By default, strings in C are null-terminated. You need to be aware of whether or not the strings you are using (if any) are null-terminated or not, and whether or not the null character (if any) is being transmitted in the message. Note that the null character is not the same as the end-of-line character. If your message formats contain any fields that are strings or sequences of characters, then you should state explicitly in your message formats whether or not you are transmitting the null

character in these fields. If you are transmitting the null character, you should account for the null character when computing the message length.

12. Both the *send* and *recv* functions return a value, which is the number of bytes read or written, or a number that is less than or equal to 0 if there is an error. My sample code in *tcpclient.c* and *tcpserver.c* does not check these returned values, but you should be checking them. **For every message sent and/or received, you should print the number of bytes sent/received as returned by these functions.**

13. If you try doing a *recv*, and the other party with whom you are communicating has broken the connection or has gone away (e.g., server being killed with a Control-C), then the *recv* will return 0. If you don't check this, your program will display a message such as "Broken pipe" at the next operation that uses the socket, and you may have a hard time debugging it. If the *send* and *recv* functions return an error value, then you should print an appropriate message and then recover gracefully if you can, particularly on the server end. This may mean closing the connection socket and going back to wait for a new connection request (on the server side), or closing the client socket and going back to open a new connection with the server (on the client side).

14. If you kill your server with a Control-C, or otherwise have an abnormal termination which does not close the server socket, and then try to immediately run the server again, you may sometimes see an error message: "can't bind to local address: Address already in use". In this case, you may have to wait for a few minutes before the port is released and becomes available for use again. You may also get this error if someone else is using the same port. If that is the reason for your error, then change your server port number.