Socket Programming

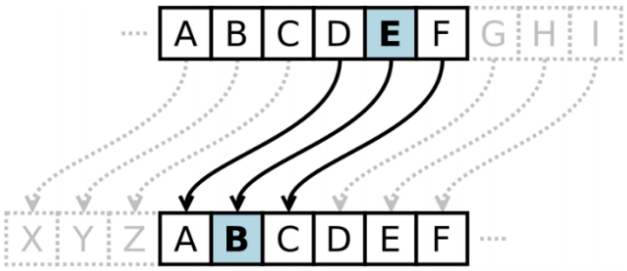
In this project, you will make a small client-server program that supports encryption. The purpose of this project is to get familiar with socket APIs because you will be implementing them in the later projects. By the end of this project, you will be able to implement connection-oriented networking programs that send data via sockets.

Important Notice
The deadline for this project is **March 18th, Thursday, 11:55 PM**. The submission link will be automatically closed after the deadline. Note that we do not take late submissions in any circumstances.

Submission link: https://forms.gle/7Pnd8EeDn22q7uxdA

What should I do?
You are asked to implement both client and server programs that communicate over TCP sockets. The client program accepts a string as a user input, and then sends the string over TCP to the server program. The server program then performs Caesar cipher on the string. The resulting string is sent back to the client, and the client prints out the result and quits. The server program should be alive for other incoming requests.

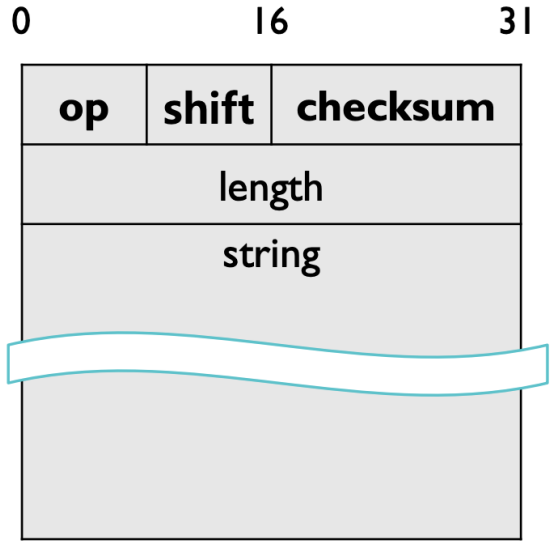
1. Caesar cipher
Caesar cipher, also known as shift cipher, is one of the simplest forms of encryption that run on strings. The algorithm is straightforward; each letter in the string is replaced with a letter corresponding to *n* letters up or down in the alphabet. For example, when *n*=3, ‘a’ is replaced with ‘d’, ‘b’ is replaced with ‘e’, and so on. (For more information on Caesar cipher, refer to https://cryptii.com/pipes/caesar-cipher.)



In this project, the given input may include both uppercase and lowercase alphabets. It may also include non-alphabetical letters, such as numbers or special characters. Your program should convert all the uppercase letters to lowercase, and perform Caesar cipher only on alphabets, ignoring other special characters or numbers.

2. Application layer protocol specification

When your client and server communicate, we need a protocol so that we can send extra data more than the string for encryption. Here, the extra data is something like, “Do we encrypt or decrypt the string?”, or “What is the value of n for encryption/decryption?”, etc.



The above is a diagram of our message structure. The numbers above the box in the diagram indicate the number of “bits”, not “bytes”.

- ‘op’ field: This field takes the first 8 bits. It denotes the operation type of the request. If this field is 0, it means encryption. If this field is 1, it means decryption.
- ‘shift’ field: This field takes the next 8 bits. It denotes the number of shifts (the value of n). The shift value is between 0~255.
- ‘checksum’ field: This field takes the next 16 bits. This is used to check whether the whole message contains errors or not. We follow the TCP checksum calculation algorithm here (1’s complement) so that you can get an idea of how TCP checksum works. For more detailed information, refer to https://locklessinc.com/articles/tcp_checksum/. **You are allowed to use the checksum implementation code provided, but make sure you cite it in the report.**
- ‘length’ field: This field takes the next 32 bits. The value of this field denotes the total length of a message, including ‘op’, ‘shift’, ‘checksum’, ‘length’ field itself, and ‘payload’(‘string’) portion. The unit of this field is in ‘bytes’, not ‘bits’. Also, this field should be in network order. (For network/host order, refer to https://en.wikipedia.org/wiki/Endianness and https://stackoverflow.com/questions/15859649/why-is-data-sent-across-the-network-converted-to-network-byte-order)
- ‘string’ field: The size of this field is variable and is determined by the value in the ‘length’ field. It contains the string we want to encrypt/decrypt.

Values on ‘op’, ‘shift’, and ‘length’ fields are treated as unsigned integers. There’s one more restriction in the protocol, that the maximum length of the message (which includes header) is

limited to 10 MB (= 10,000,000 bytes). See https://www.quora.com/What-is-the-difference-between-a-megabyte-and-a-mebibyte).

3. Client specification

- Your client program should be able to take an IP address, a port number of a server and operation type and shift through the command line parameters. Example usage of the client program should look like this:
  $ ./client -h 143.248.111.222 -p 1234 -o 0 -s 5
  In this case, the server's IP address is 143.248.111.222, the server's port number is 1234, and we are encrypting with n=5. You must follow this command line parameters format and binary name (which is 'client').
- The client takes the string by standard input (stdin). The client then creates the message and sends it to the server. Each data should be wrapped in the correct message format that follows the protocol above. Once the client receives a reply from the server, it should unwrap the message and only print the resulting string to the standard output (stdout). Note that if the input is too long (more than 10 MB), then the client should split the input and wrap it in different messages. Otherwise, the server will reject the request.
- Your client program should be able to handle any kind of string, including binary data.
- Your client program should only terminate when EOF (end-of-file) is received in stdin.

4. Server specification

You should also implement a server program that can communicate with the client program you have built above.

- Your server should receive the required information through the command line parameters. An example usage of the server program should look like this:
  $ ./server -p 1234
  Unlike the client, when you run your server, you should be able to set the server's port number manually. In fact, this is very natural that by setting the server's port number with the desired value we can advertise the port number to the clients out there. You must follow this command line parameters format and binary name (which is 'server').
- The server program listens to incoming connection requests and then accepts them. Once the server program receives a string from the client, the server performs Caesar cipher on this string and sends the resulting message back to the client.
- Your server program must be able to handle multiple connections in parallel. You have several options for this requirement, like fork() or select()/epoll(). Refer to http://alas.matf.bg.ac.rs/manuals/lspe/snode=93.html
- Your server should reject connections from clients that violate the protocol.

5. Extra specification

- Write your program in C. No other languages are allowed.
- Only use C standard libraries and Linux system calls. No other 3rd party libraries are allowed.

- When you send the data, it should follow the network byte ordering. Refer to the following document if you are not familiar with endianness.
  https://en.wikipedia.org/wiki/Endianness
- You should also provide Makefile to compile your code. We will type
  $ make all
  in your submission. This will be done by automated scripts. When this '$ make all' is issued, your Makefile should generate two executable files, 'client' and 'server'.
- Write a report about your work. Briefly explain your implementation. Cite all references if you received any help. It should be no more than 3 pages, and it should be in **PDF.**

6. Guide to socket programming
Go to this guide (https://beej.us/guide/bgnet/html/multi/index.html) and read what kind of APIs are out there. There are many example codes as well. **You are allowed to use code segments and socket API usages in the guide above, but make sure you cite it in the report.**

7. Tips
- Please be aware that a message in our protocol is not served by a single network packet. As you implement an application-layer protocol on top of TCP (like HTTP), your message can be split into multiple packets no matter how small your message is.
- Your client program should print nothing but encryption/decryption results. Please avoid printing '\n' at the end of enc/dec results to make it look better on the Terminal. Use stderr if you need to print something else for debugging purposes.

How do I test my implementation?
We provide you a few sample input/output files for your program. However, when grading, we will use a different set of inputs (which are typically longer and more complicated).
Sample files: https://drive.google.com/file/d/1kkWCUMJl8PQDYG_ttzLGy3AYnTo5tfxP
Sample input/output files use shift = 5.

Also, we opened four different test servers for your debugging and testing. The detailed information on the test servers is:
- Test server 1
  - eelab5.kaist.ac.kr:12000
  - A reference implementation of the server in our project. You can test your client with this server.
- Test server 2
  - eelab5.kaist.ac.kr:12001
  - This server does not check the validity of checksums of incoming messages. This might be especially useful if it looks like test server 1 randomly ignores your request message (more precisely, cuts a connection) when message size is larger than 64 KB (if you experience this, you might want to check the FAQ section at the bottom of this document).
- Test server 3

    - eelab5.kaist.ac.kr:12002
    - Response messages from this server always have intentionally wrong checksum values. As soon as you fail at checksum validation, your client should cut a corresponding TCP connection.
- Test server 4
    - eelab5.kaist.ac.kr:12003
    - This server is a laggy one. It works correctly, but you might need to call read() repeatedly to read a short message. Please be aware that even our 8-bytes header might not be fully read by a single read() call. Your client should experience no problem with this server.

Be careful not to set the same port with the test server. Also, start the project early; if many clients attempt to connect to the test server, you might be dropped.

We will run your submission in the following environment (same as the Haedong server):
Ubuntu 17.10 64-bit, Linux 4.13.0-46-generic, gcc 5.4.1
Note that any problem that is caused by using different environments is the student's responsibility. We will not accept any regrade request such as “It runs on my computer”.

What is the grading criteria?

This project is worth **4.5%** of your total grade. The grading will be done as below.

- (30%) Code: Client implementation
- (50%) Code: Server implementation
- (20%) Code: Makefile
- (0%) Report: The report will not be explicitly graded to the numerical value, but we will use it when there are any issues with grading.
- Any violations (wrong Makefile script, wrong file name, report is not PDF, etc.) will result in a penalty.
- If your code has formatting problems (such as adding unnecessary \n or other whitespaces to make output pretty), we will deduct 20% from your grade that you will get when the formatting problem is solved. As mentioned above, don’t add anything but just simply print out the encryption/decryption result.
- We will run your submission 10 times and will give you the lowest score.
- We will set a 30 seconds limit per test case first, and re-run those with a 3 minutes limit that failed due to the timeout on the first trial. (There will be no penalty for failing a 30 seconds test but passing a 3 minutes test.) If your code fails to run in 3 minutes, you will get no credits for that test. Since we will run your programs on the same machine, they will typically run faster than the case where your program “actually” communicates with our reference server.

Where can I ask questions?

We encourage students to actively participate in the course. Please share your experiences and questions in the course Campuswire group. However, if you think your question is not appropriate for sharing for some reason (such as it contains private information), send it to

ee323@nmsl.kaist.ac.kr. However, TAs may share your question on Campuswire if we find it appropriate.

How to submit?
You need to submit the following items. Note that in this project, we don't provide any skeleton files.

- client.c
- server.c
- Makefile: '$ make all' should generate executable files 'client' and 'server'
- report.pdf

You can add extra c/h files, but make sure to properly build in Makefile and add explanations of extra files at the report.
Compress the above items into one zip file. Name it as **{studentID}_{name(in English)}_project1.zip** (e.g. 20211234_HyungJunYoon_project1.zip) and submit it to the link below.

Submission link: https://forms.gle/7Pnd8EeDn22q7uxdA

Q&A
Q. How much implementation is required in this project?
A. My implementation has 277 lines for client.c, and 292 lines for server.c. However, if you are not familiar with sockets, you will spend a lot of time debugging with socket APIs. This could be very difficult sometimes, so please start early.

Q. TCP/IP checksum algorithm seems pretty clear. But, their range is somewhat vague. Is data range restricted to payload only or they include op, shift, and total length?
A. In TCP/IP RFC (RFC 793), the checksum field is defined as follows.
"The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header and text"
Our checksum field is exactly the same. It should be calculated for all header fields (op, shift, length, and data).

Q. Does the 10 MB limit include header (op, shift, checksum, length = 8 Bytes, thus the maximum size of the string is 10 MB - 8 B) or not?
A. 10 MB includes the header. (Tip: the value of the length field includes the size of the header.)

Q. When should the client send a message to the server? Like, when they meet a new line from the input string, the input string has accumulated up to 9984 Bytes, etc.
A. The way you split the string received from stdin is up to you. Just make sure the size of a message does not exceed 10 MB.

Q. In client, is it ok to assume input is “printable string + ‘\n’” * many times? Specifically, does the input file consist of (([A-Za-z0-9!"#$%&'()*+,.\\/:;<=>?@^_`{|}~\t-])*\n)+ in regular expression?
A. Consider an input file is a regular text file. And the file has an EOF at the end. When you redirect a text file to stdin (i.e. ./client < sample.txt), EOF is also streamed into stdin. When you write a string to stdin, you can signal EOF with ctrl+D in the Linux system. (Tip: Whatever function you use for reading a text from stdin, checking whether the input string has EOF would be helpful.)

Q. Can we assume that the server's IP is IPv4?
A. Yes.

Q. Why do we use big endian for each field? Shouldn’t the entire message be big endian?
A. Before answering the question, I want you to be clear on the concept of a network protocol header and endian.
On one hand, the network protocol header, such as our {op, shift, checksum, length}, specifies how we should interpret each part (field) of a message. For example, our header specifies that the very first byte represents the kind of operation to be performed. On the other hand, endian specifies which byte is the least significant byte (LSB) and which byte is the most significant byte (MSB) in a multi-byte variable. For example, big-endian specifies that the first byte is MSB. In summary, the network protocol is for a message and endian is for a multi-byte variable.

Q. Why should we not use ntohs() or htons() for checksum?
A. The checksum is for error checking, and its mathematical value is not important. (In addition, checksum calculation does not care endian due to wrap-around logic).

Q. The buffer of the server should be at least 10 MB so I declared it like “char buf[10*1024*1024];”. But I think it can exceed memory capacity when many processes are created at the same time. Do we need to manage memory using dynamic allocation?
A. You should dynamically allocate the buffer. As you may have learned in previous courses, if you declare a static array then it is allocated in stack. Typically, Linux kernel has a default stack size limit for each process and it was 8 MB for me. So a 10 MB buffer should be allocated in heap. You can check your own stack size limit by “ulimit -s” command which returns the stack size limit in KiB.

Q. My client works fine with short messages, however, the test server cuts connections when the message size is large (several megabytes).
A. For a TCP packet, the packet size (more specifically, IP fragment size) cannot exceed 65535 Bytes. (And the typical maximum packet size is 1514). Most TCP checksum calculation codes on the Internet assume this. They usually calculate the checksum in three separate steps, 1. Summation, 2. Wrap-around, 3. Inversion. In the summation step, they use a 32-bit variable (unsigned, unsigned int, or uint32_t). This is okay since 32 bits (precisely, 31 bits) are sufficient for summing any 32768 (= 65535 / 2) unsigned 16-bit variables. However, in our case, the

message size can be 10 MB which is much larger than 65535. This demands you to update your checksum implementation.

Q. I was struggling with the checksum field, since I considered it as a big-endian. Now I know that the test server will reply to my messages only if the checksum is not packed by htons, but still I can't reason why it works. Why does the checksum field not care about endian-ness, even if it is 2-bytes long?
A. You may want to read RFC 1071, Computing the Internet Checksum.
See (B) Byte Order Independence on page 3. Link: https://tools.ietf.org/html/rfc1071

Q. Should the client validate checksum and the length part of the response from the server?
A. Yes, the client should.

Q. Can we assume that input strings will be ASCII texts? ie 1byte encodings?
A. You can just consider the strings on the given test vectors.

Q. When should the server terminate? Does it just loop forever?
A. Yes, it lasts until the user intentionally stops it.

Q. What is the protocol for handling invalid parameters? Do I shut down the program? Or should I ask the user to input those inputs again?
A. Shut down the process for invalid parameters. no output should come out from it.

Q. Do we have a specific format for stdout outputs for server and client?
Can we have informative messages like "connecting to : 127.0.0.0" or "terminating - client disconnected," for example?
A. For stdout,

1. Your client should never print any message except the result of encryption/decryption.
2. Your server should not print anything.

You can use stderr for your own debugging messages.

Q. I wrote some codes in files 'protocol.c/h', which is not in the submission format. But will it affect the evaluation process even if I handle it with proper Makefile?
A. It is okay if

1. Your Makefile works fine
2. Additional files (for your case, protocol.c/h) are written by you or have proper references.

Splitting your source code into multiple files is a good habit.

Q. How can I test my server with multiple clients? I can do it with one or two clients by opening more shells, but I am not sure. Is there any efficient way to test my server with multiple clients?
A. Maybe trying a simple shell script will work. [How to run multiple processes in bash]
http://stackoverflow.com/questions/5238103/how-to-startmultiple-processes-in-bash

Q. Whenever I try to receive something from the test server, I get “Connection reset by peer”. What should I do?
A. We programmed the server to cut the connection when it detects any violation of the protocol (and you should also do the same with your server). Please check whether your client makes legitimate request messages.

Q. When I calculate a checksum if the length of data is an odd number, should I put '0' to the end of data to make 16-bit pieces? For example, when data is 'aabbbcc', should I split it into 'aa' 'bb' 'bc' and ‘c0’?
A. Yes, you should add additional zero bits at the end of the data to make the data into 16-bits pieces.

Q. If input data's size is over 10MB, do I have to split input data under 10MB each, and make requests more than 1 time, and finally merge all of the server's responses and print it to the standard output?
A. That is generally correct, but how you split data and merge responses is up to you.

Q. Do we set a default port# for the server if not given with the argument? Or should we only accept the format with one?
A. Latter is correct. You should only accept the format with the right number of variables.

Q. How can the server know the client is disconnected? I want to close the child process if the client is disconnected. But the client does not do anything when close connected fd.
A. The first read()/write() call after the peer's close() call returns zero. (if you use a Linux based operating system, we strongly recommend using the man command or pages to check the definition and details of a function)

Q. I am doing my project outside the campus and cannot seem to connect to the server. What should I do?
A. If you are doing your project outside campus (including Eoeun-dong, Gung-dong, etc) you should use KVPN to connect to the server. The link is as follows:
https://kvpn.kaist.ac.kr/dana-…/auth/url_default/welcome.cgi