COMP 2401B -- Assignment #4

Due: Thursday, December 6, 2018 at 12:00 pm (noon)

Collaboration: You may work in groups of no more than two (2) students

Goal

Timmy Tortoise and Harold the Hare are still being held captive by an evil wizard who wants to use their technical skills to help him get revenge on his rivals. Our heroes were hatching an escape plan, but the wizard found them out! He has retaliated by locking up our heroes in separate cells. Luckily, Timmy managed to stash away some of the wizard's communications equipment and has a plan to use it for sending secret messages to Harold. Using the VM provided for the course and the socket TCP/IP code we saw in class, you will assist Timmy by implementing an encrypted chat utility. Your program will help Timmy and Harold send encrypted messages to each other that the evil wizard is unable to decrypt.

Learning Outcomes

You will practice problem solving and designing modular functions to implement a solution. You will implement a peer-to-peer system that communicates over TCP/IP sockets. You will also be integrating into your code a function that is provided as object code only.

Instructions

1. From client-server to peer-to-peer

Your program will consist of a **single executable** that combines the client and server code that we saw in class. When your executable starts up, it will check to see if there was a command line argument entered after the executable name. If there is no command line argument, the program will wait for a connection request to come in, as if it were a server waiting for a client to contact it. If there is a command line argument, it will be the IP address that the program needs to connect to. In that case, it will initiate a connection request to that IP address, as it it were a client connecting to a server.

When you run your program, you will launch the first instance of it in one window, using no command line argument. That program will go into wait mode. In another window, you will launch a second instance of the same program, using the home IP address 127.0.0.1 as a command line argument (in principle, this could be any IP address in the world, but we'll keep it simple so that we don't have deal with firewalls). Once both programs are connected to each other, the encrypted chatting can begin.

2. Design your program

The first step is to break down your program into modular and reusable functions. You must determine which functions you need to implement, given the TCP/IP code that we used in class. You need to blend that client-server code into a single program, and it has to be made up of modular functions.

You are also required to divided up your code into separate source files and provide a header file. Do **not** put all the code in the `main()` function, and do **not** put it all in the same source file. You will also provide a Makefile to facilitate program building.

3. Encrypted chat utility

Once two instances of the program have established a connection, as explained in Instruction #1, the program will go into *chat mode*. Chat mode works as follows:

- the users will take turns; at each turn:
  - the user whose turn it is will be prompted for a message; this message can be multiple words long
  - the message will be encrypted using the algorithm described below
  - the encrypted message will be sent over TCP/IP to the other user
  - the receiving program instance will decrypt the message (using the same algorithm as encryption), and it will display both the encrypted message and the decrypted equivalent to its user
  - the receiving user will then take their turn
- the program instance that initiated the connection (the one behaving as client) will be first to prompt its user to enter a message
- the chat ends when one of the users enters “quit” as their message; when this happens:
  - the “quit” message is encrypted and sent to the other user
  - both program instances end

NOTE: It’s recommended that you do *not* use the `strlen()` function to compute the length of an encrypted message. The encrypted messages sometimes contain zeros within them, which the `strlen()` function will treat as a null terminator. This can result in the wrong number of bytes being decrypted or printed to the screen, and the encryption counters in the two program instances will get desynchronized. Instead, you should use the return value of the `recv()` system call, which indicates the number of bytes that were actually received.

A sample execution is shown in Figure 1.

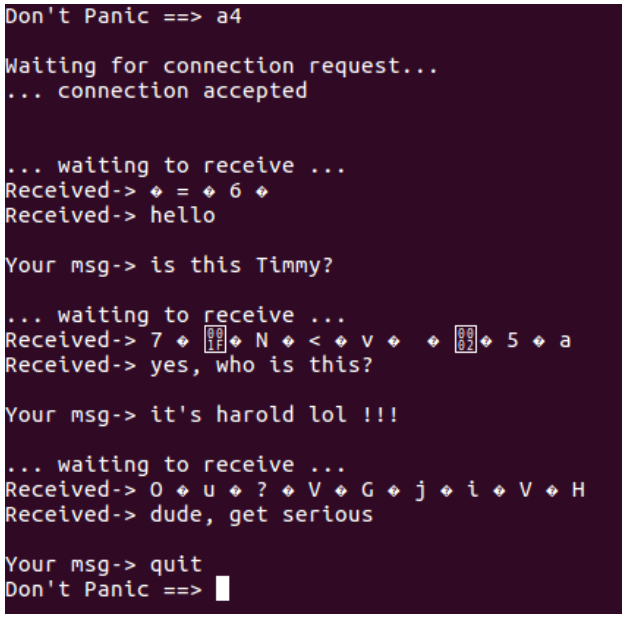

Figure 1: Sample execution

4. Encryption and decryption of messages

You will download the `a4Posted.tar` file from *cuLearn* and un-tar it in your own VM. This archive contains a single file: `a4-util.o` which is an object file that you will link with your own code in the Makefile that you provide. This file provides the `encrypt()` function, which has the following prototype:

```
unsigned char encrypt(unsigned char c, unsigned char k)
```

It encrypts the given character `c` with the given key `k`, and returns the encrypted character as the return value.

You will use the following encryption/decryption algorithm, which is based on the Cipher Block Chaining technique in Counter mode (CBC-CTR):

- you will declare two global variables as unsigned chars: one variable for the encryption/decryption *key*, and one variable for the *counter*
  - we'll use Timmy's favourite numbers for initialization: you will initialize the key to 101 and the counter to 87
- every time the program sends a message, it will encrypt the plain text message into its ciphertext equivalent before transmission:
  - the program will loop over every character of the plain text message; for every character:
    - the global counter is encrypted with the global key using the provided `encrypt()` function
    - the encrypted counter is `xor`'d with the plain text character to produce the corresponding ciphertext character (the `xor` operator in C is `^` , as in the expression `a^b`)
    - the global counter is incremented by 1
  - once the entire plain text message has been encrypted into its ciphertext equivalent, the cipher text is sent to the receiving program
- every time the program receives a message, it will decrypt the received ciphertext into its plain text equivalent using the exact same algorithm as for encryption. It does **not** perform the steps in reverse order! The same steps in the same order will work, as long as the same key and counter values are used. For example, if the string "hello" is encrypted with the counter beginning at value 87 and ending at 91, it must be decrypted using a counter with those same values.

Constraints

- your program must be correctly designed and separated into modular, reusable functions
- your program must reuse functions everywhere possible
- your program must perform all basic error checking
- your program must be thoroughly documented, including each function and parameter
- compound data types must always be passed by reference
- all dynamically allocated memory must be explicitly deallocated
- do not use any global variables, except where explicitly permitted

Submission

You will submit in *cuLearn*, before the due date and time, one `tar` or `zip` file that includes the following:

- all source code, including the code provided, if applicable
- a Makefile
- a readme file that includes:
  - a preamble (program author, purpose, list of source/header/data files)
  - the exact compilation command
  - launching and operating instructions

If you are working with a partner:

- only **one partner** submits the assignment, and the other partner submits **nothing**
- the submitting partner must enter *the names of both partners* in the **Online Text** box of the *cuLearn* submission link
- the readme file **must** contain the names of both partners

Grading (out of 100)

Marking components:

- 20 marks: correct connection sequence, based on command line argument
- 40 marks: correct implementation of the main chat loop
- 40 marks: correct implementation of encryption/decryption algorithm

Deductions:

- Packaging errors:
  - 100 marks for an incorrect archive type that is not supported by the VM
  - 50 marks for an incorrect archive type that is supported by the VM
  - 10 marks for missing readme
  - 20 marks for missing Makefile
  - 10 marks for not separating the code into different source files
- Major programming and design errors:
  - 50% of a marking component that uses global variables, unless otherwise permitted
  - 50% of a marking component that is incorrectly designed
  - 50% of a marking component that doesn't pass compound data types by reference
  - 100% of a marking component where the function prototype has been modified, where applicable
- Minor programming and design errors:
  - 10 marks for consistently missing comments or other bad style
  - 10 marks for consistently failing to perform basic error checking
  - up to 10 marks for memory leaks
- Execution errors:
  - 100% of a marking component that cannot be tested because it doesn't compile or execute in VM
  - 100% of a marking component that cannot be tested because it's not used in the code
  - 100% of a marking component that cannot be proven to run successfully due to missing output