15-112: Fundamentals of Programming and Computer Science, Fall 2019

Homework 7 Programming: Sockets Programming and Authentication

Due: Tuesday, October 29, 2019 by 22:00

This programming homework is designed to get you more practice with communicating over the Internet with a server. Although you will submit your final code on Autolab, your work will be graded after you finish the homework. We will also grade you on the overall structure of your code to make sure you make good design decisions. Download *Client.py* from course website under assignement 7, rename the file as YourAndrewIDhw7.py. **Make sure you read this file** and understand how the code is structured. Write all the missing functions in the same file (You are encouraged to also create your own helper functions) and then submit this python file under lab7 option at:

`https://autolab.andrew.cmu.edu/courses/15112q-f19`

1 Getting Started

As you probably know, different programs can communicate with each other over networks. This allows computers to share resources and services. In one very popular networking model, we call the program that requests the service as a client and the program which provides the service as a server. For example, when you check your bank account from your computer, a client program in your computer forwards a request to a server program at the bank. Computer transactions in which the server fulfills a request made by a client are very common and the client/server model has become one of the central ideas of network computing.

In this homework, you will be implementing a client program that communicates with a server to create a simple chat service. The client connects to the server using the socket interface. There is some starter code available, you will fill in the missing functions.To start this homework, you will need to connect and communicate with a server.
Your program will work in the following manner:

- Your client code will establish a socket connection with the server.
- You will login to the server using your username and password - by default, your user name and password are the same as your andrewID.
- Once logged in, you will use the functions you implement to send and receive friend requests, messages and files from other clients.

- You will write a different function for each task in order to organize your code better. You should pass the socket connection to this function and not create a new connection every time.
- You will be graded on your coding style for this assignment. You can always ask your instructors for coding style guidelines.

Task 0. To start this homework, you will need to connect and communicate with a server. Fill in the function *StartConnection* that takes an IP Address and port number as input parameters and returns a socket connection. The server for this homework is at IP Address 86.36.46.10 and is listening on port number 15112.

2 Knock knock!

In this task you will write code to login to the server. You already know your username and password. You will use the following protocol for login:

- Send the command "LOGIN username\n" to the server.
- The server will respond back with "LOGIN username CHALLENGE" where CHALLENGE is a string you will use to calculate your messagedigest"
- Send the command "LOGIN username messagedigest\n" to the server.
- The server will response back with either success or failure message. If you are successful in logging in, you can sending commands to the server.

We will be using a variant of the popular message digest called MD5.The messagedigest can be calculated in the following way:

- Lets say your password is PD = "p1p2p3p4p5...pn" where p1 is the first character, p2 is the second character and so on.
- Lets say the challenge string is CH = 'c1c2c3c3....cm" where c1 is the first character, c2 is the second character and so on.
- n is size of password and m is size of challenge string. There is no relationship between n and m.
- Create a string called message which is a concatenation of PD and CH in this order
- Create a block that is 512 characters long. The first n+m characters of this block are the same as message created in the previous step. The last three characters of the block determine the size of the string which is equal to n+m. The missing character in the middle are filled in with character 1 followed by enough 0's to make block 512 characters long. For example, let's say password is srazak and challenge received from the server is ABCD123abcd. The block will look like the following:

```
"srazakABCD123abcd1000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000017"
```

The last three characters "017", represent the size of the string "srazakABCD123abcd"

- Break the above block into sixteen 32-character chunks. Find the sum of ASCII values of each character of each chunk and save it in M such that M is list of 16 integers. M[j], 0 <= j <= 15, where M[0] is Sum of ASCII characters of first 32 characters of the above block, M[1] is the sum of ASCII values of next 32 characters and so on. For the example above, the values of these sixteen chuncks are:

```
M[0] = 2183
M[1] = 1536
M[2] = 1536
M[3] = 1536
M[4] = 1536
M[5] = 1536
M[6] = 1536
M[7] = 1536
M[8] = 1536
M[9] = 1536
M[10] = 1536
M[11] = 1536
M[12] = 1536
M[13] = 1536
M[14] = 1536
M[15] = 1544
```

- Process each chunk following the pseudocode given below (adapted from pseudocode given at `http://en.wikipedia.org/wiki/MD5#Pseudocode`): s and K are lists with 64 values shown below.

```
s[0..15] := {7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22}
s[16..31] := {5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20}
s[32..47] := {4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23}
s[48..63] := { 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21 }

K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
```

```
    K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
    K[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
    K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
    K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
    K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
    K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
    K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
    K[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
    K[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
    K[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
    K[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
    K[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
    K[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
    K[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
    K[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }

    //Initialize variables:
    var int a0 := 0x67452301   //A
    var int b0 := 0xefcdab89   //B
    var int c0 := 0x98badcfe   //C
    var int d0 := 0x10325476   //D
    var int A := a0
    var int B := b0
    var int C := c0
    var int D := d0
//Main loop:
    for i from 0 to 63
        if 0 <= i <= 15 then
            F := (B and C) or ((not B) and D)
            F := F & 0xFFFFFFFF
            g := i
        else if 16 <= i<= 31
            F := (D and B) or ((not D) and C)
            F := F & 0xFFFFFFFF
            g := (5xi + 1) mod 16
        else if 32 <= i <= 47
            F := B xor C xor D
            F := F & 0xFFFFFFFF
            g := (3xi + 5) mod 16
        else if 48 <= i <= 63
            F := C xor (B or (not D))
            F := F & 0xFFFFFFFF
            g := (7xi) mod 16
        dTemp := D
        D := C
```

```
        C := B
        B := B + leftrotate((A + F + K[i] + M[g]), s[i])
        B := B & 0xFFFFFFFF
        A := dTemp
    end for
//Add this chunk's hash to result so far:
    a0 := (a0 + A) & 0xFFFFFFFF
    b0 := (b0 + B) & 0xFFFFFFFF
    c0 := (c0 + C) & 0xFFFFFFFF
    d0 := (d0 + D) & 0xFFFFFFFF

result := a0 append b0 append c0 append d0


//leftrotate function definition
leftrotate (x, c)
    return (x << c)&0xFFFFFFFF or (x >> (32-c)&0x7FFFFFFF>>(32-c));

and, or, not are bit-wise operators.
```

- The result variable calculated above is a string that is the final message digest.

  This algorithm is hard to debug since the only way for you to know that it does not work is if the server rejects the hash value you sent. To make the debugging task easier, use the following values while debugging your MD5 hash code:

  Let's assume that your password is "passwaard" and the challenge sent by the server is "ABCDEFGHIJKLMNOP".

  The 512-character Block will be as follows:

```
passwaardABCDEFGHIJKLMNOP1000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
00000000000000000025
```

  Using the above block, the value of 16 integers in the list M will be:

```
2463
1536
```

```
1536
1536
1536
1536
1536
1536
1536
1536
1536
1536
1536
1536
1536
1543
```

At the end of the first iteration of the "Main" loop, before variable 'A' is assigned the value of dTemp, the values of different variables should be:

```
at i =  0 :
A= 1732584193
B= 2770646708
C= 4023233417
D= 2562383102
F= 2562383102
g= 0
dtemp= 271733878
```

Task 1. *(10 points)* Write the function *login* that takes a socket connection, a username as a string and the password as a string then applies the protocol described above to log this user into the server. The function should return true if the logging in process was successful and false otherwise.

3 Who's there?

Now that you're logged into the chat service, you can start using the services provided by the chat server. If you type Menu, you will see a list of all possible services:

- Menu: Shows a Menu of acceptable commands
- Users: List all active users
- Friends: Show your current friends
- Add Friend: Send another friend a friend request
- Send Message: Send a message to a friend

- Send File: Send a file to a friend
- Requests: See your friend requests
- Messages: See the new messages you received
- Exit: Exits the chat client

Currently, however, the chat does not communicate with the server and it simply returns trivial answers to each of these commands. Your job is to study the protocol described below, then use it to complete the implementation of the chat client. A protocol is a set of communication rules shared between any two computers that defines the messages that they can exchange and what each message means.
size is the size of the string being sent from beginning to end including the size field. *size* should always be 5 digits long. (not including the null terminator). Notice that every time a message is correctly sent to the server, the client must receive some reply.

Message from Client (Your Code)	Reply From Server
@users	@size@users@n@[$user_1$]@[$user_2$]@....@[$user_n$]
@friends	@size@friends@n@[$friend_1$]@[$friend_2$]@ ....@[$friend_n$]
@size@request@friend@[username]	@ok
@size@accept@friend@[username]	@ok or @no such friend request
@size@sendmsg@[username]@[messageText]	@ok
@size@sendfile@[username]@[filename] @[fileContent]	@ok
@rxrqst	@size@n@[$user_1$]@[$user_2$]@...@[$user_n$]
@rxmsg	@size@n@msg@[$user_1$]@[$message_1$]@msg@ [$user_2$]@[$message_2$]...@file@[$user_m$] @[$filename_m$]@[$fileContent_m$]@file@[$user_{m+1}$] @[$filename_{m+1}$]@[$fileContent_{m+1}$].... @file@[$user_n$]@[$filename_n$]@[$fileContent_n$]

Task 2. *(10 points)* Write the functions *getUsers*(@users), *getFriends*(@friends) and *getRequests*(@rxrqst) that each takes a socket connection as input parameter and returns a list of usernames. The first should return a list of active users, the second should return a list of users who you have as friends and the third should return a list of users who requested to become your friends. Keep in mind that the message sent by server might be longer than

the number of bytes you are reading. Make sure you check the "size" field in each message to confirm you have read the message completely.

Task 3. *(10 points)* Write the functions *sendFriendRequest*(@*request*@*friend*) and *acceptFriendRequest*(@*accept*@*friend*) that each takes a socket connection and a username as input parameters and carries out the correct task *sendFriendRequest* sends a friend request to the server and *acceptFriendRequest* sends a friend request acceptance to the server). Both functions should return true if the task was carried out successfully and false otherwise.

Task 4. *(5 points)* Write the function *sendMessage*(@sendmsg) that takes a socket connection, a username and a string as parameters and sends the string (message) to the user. This function should return true if the message is sent succesfully and false otherwise.

Task 5. *(5 points)* Write the function *sendFile*(@sendfile) that takes a socket connection, a username and a filename as input parameters. This function should read the file and send it in the correct format to the user identified. It should then return true if the file was sent correctly and false otherwise.

Task 6. *(10 points)* Write the function *getMail*(@rxmsg) that takes a socket connection and returns a tuple with two lists: list of tuples (user, message) representing all received messages and a list of tuples (user, filename) representing all files received. Your function should also save all the files that it receives to local directory under the filename it got.

Task 7. *(0 points)* In this assignment, you will be not be explicitly graded on style. You should, in general, follow the same guidelines as earlier assignments. Following are some suggestions on how to make your code look better:

- Make sure your code is properly commented
- Make sure you use variable names that represent data being stored in these variables
- Put your name and andrewID on the top of each file as a comment.
- You are required to break your tasks into appropriate functions and call those functions from within your program. Make sure you don't make functions for the sake of making functions but have some logical division of tasks. Before each function, have a block of comments that explains what this function does, what each input means, any restrictions on the input (for example: x should be a prime number), and what return value, if any, to expect from the function.