**Operating Systems Spring 2017**

**Messaging Server (Final Project)**

You will develop a server whose basic functionality is to forward messages to a group. Clients that use this server can both register to listen to messages and they can send messages.

As always, any assistance/code you get in your solution from the internet or your colleagues must be identified, except that you may use any code from me freely. The entire project with report is due on Friday April 21 at midnight, but start work right away, because the first 2 parts have their own deadlines – part 1 on April 10 and part 2 on April 15.

Because there is a large amount of functionality, the work is separated into 3 parts in order to help you be as successful as possible. You will submit each part individually, with the idea that you will get one part working before ***adding*** the functionality of the next part. If you try to do it all at once, you will cause yourself trouble. I also advise you to make good use of functions and multiple C files to make your code more readable and easy to debug and extend. You should plan on doing one part per week, approximately (first part a little longer than the next two parts).

**Part 1: the basic server and client**

This server should be written as a multiplexing server, which should use threads only in order to handle long requests. Basically the server will get messages from clients, and forward those messages to other clients who have registered some interest in them. The server does not save messages, only forwards them to clients who are currently online. Clients may register to receive all messages, or register to receive just certain messages tagged with some identifier. A tag is assumed to be a single word (no spaces), and anything else at the end of the command should be ignored.

The server does not respond to any of the commands from the client. If the client deregisters from a tag it was not registered for, the server just ignores it. Tagging a message is optional for the client, so to indicate a tag, the **‘#’** symbol (as in hashtag) is used. If the message starts with **‘#’** then the first word is assumed to be the hashtag, and the rest of the packet is the message. If there is no hashtag, then that message will only go to clients who have sent the REGISTERALL command. A client can register interest in any number of tags. When a message comes with a tag, it must be forwarded to all clients who have registered interest in it. You may set a maximum value for the number of tags the server can manage, but the server should never track tags for which no one is currently registered (i.e., once no client is registered to receive messages with some certain tag, that tag should disappear from the server’s “memory”).

Messages are assumed to be simple single-line messages, so a newline (‘\n’) indicates the end of a message. A message may only have a single hashtag. No correct message from a client (the entire line starting from the word MSG and including the trailing newline) will exceed 2048 bytes.

**Protocol**

|  |  |
| --- | --- |
| **Client sends** | **Server action** |
| **REGISTERALL** | From now on, server forwards all messages including those sent from this client itself to this client. Client becomes a listener. |
| **DEREGISTERALL** | From now on, server does not forward any messages to this client. *This revokes the registerall command as well as all individual tag registrations as well.* |
| **REGISTER *tag*** | Forward all messages tagged with *tag* to this client. |
| **DEREGISTER *tag*** | Stop forwarding all messages tagged with *tag* to this client. |
| **MSG *message*** | Server takes this message and sends it to all registered listeners. |
| **MSG [#*tag*] *message*** | Optionally tag a message with *tag*. |

You should write a simple client that allows you to test your server. The simplest client might just allow the user to type the messages at the command-line and displays the messages that arrive. Your client should also allow for duplicating connections to put load the server.

**Protocol**

|  |  |
| --- | --- |
| **Server sends** | **Client action** |
| **MSG *message*** | Client shows the message |
| **MSG [#*tag*] *message*** | Client show the message with the *tag*. |

**Part 2: send an encrypted message**

Extend your server to include the ability to forward encrypted messages. This requires a special message, because the message size must be sent before the encrypted text, since it will not be possible to rely on a newline terminator. Your client must be extended to encrypt/decrypt the messages. The server is not concerned with this process, it just forwards messages. Forward encrypted messages using the same guidelines as for the basic server. Tags are not encrypted.

Your client will use a simple encryption algorithm called RC4, which can encrypt a stream of bytes. The RC4 algorithm requires a password (key) for the encryption/decryption process. We will assume that clients exchange the key across some other channel (voice, email, etc.). Your client program must allow the key to be entered (or replaced) at any time while the client is running. Setting (or resetting) the key starts (restarts) the encryption/decryption mechanism. Here is a website with C code for the RC4 algorithm:

* <http://bradconte.com/rc4_c>

The developer of this code provides a good explanation. The **ksa()** function initializes the data structure needed to generate bytes of ciphertext using the specified key. The **prga()** function produces a continuous stream called a keystream, a given number of bytes at a time. To encrypt, XOR each plaintext byte with a keystream byte, and to decrypt, XOR an encrypted byte with a keystream byte. It can be understood that this algorithm is symmetric, so encryption and decryption are actually the same.

**Protocol - server**

|  |  |
| --- | --- |
| **Client sends** | **Server action** |
| **MSGE *bytecount/msgbytes*** | Server forwards this apparently encrypted message to all registered listeners. |
| **MSGE [#*tag*] *bytecount/msgbytes*** | Optionally tag an encrypted message with *tag*. |

**Protocol - client**

|  |  |
| --- | --- |
| **Server sends** | **Client action** |
| **MSGE *bytecount/msgbytes*** | Client decrypts and shows the message |
| **MSGE [#*tag*] *bytecount/msgbytes*** | Client decrypts and shows the message with the *tag*. |

**Part 3: send an image**

Extend your server to include the ability to send an image (with or without a tag). Forward images using the same guidelines as for the basic server. The image size must be sent before the image, so that all the bytes can be read. I will provide you a client for testing this part of the work.

**Protocol**

|  |  |
| --- | --- |
| **Client sends** | **Server action** |
| **IMAGE *bytecount/imagebytes*** | Server forwards this image to all registered listeners. |
| **IMAGE [#*tag*] *bytecount/imagebytes*** | Optionally tag an image with *tag*. |

**Technical Report**

Along with your project, you will submit a technical report, describing the structure and design of your server. A clearly-specified design can, theoretically, make up (in your score) for some areas in the code that you couldn’t get working. Computer code is, essentially, a specification of something you want the computer to do. Sometimes, bugs stop you from reaching the goal of a completely working program in the timeframe of homework, but if the person grading can understand your (correct) intention, you can get more points.