CMPUT 379, Assignment 2, Fall 2016

(IPC, Synchronization, Message Passing, Sockets)

Objective

You are asked to implement a chat server and its accompanying client. Developing the solution requires that you get familiar with programming using the sockets API, which will be covered in the labs. The specifications of the assignment include a specification of the protocol between client and server and a minimum of client-side and server-side behavior/interaction.

Protocol Specification

Initial Handshake

Upon connection of a client to the server, the server responds by sending the following two bytes (and in this order, from left to right) shown in hexadecimal:

`0xCF 0xA7`

The client interprets reading those two bytes as a confirmation that it has connected to a compatible server. This sequence is immediately followed by a 16-bit unsigned integer number sent by the server in *network byte order* specifying the number of users currently connected to the server, followed in turn by the list of their user names. A single user name is sent as a record of the form:

length string

where *length* is a single (unsigned) byte indicating the length of the *string* that follows it, and *string* is simply the sequence of bytes (characters) of the indicated *length*. Note that the *string* does not contain the terminating `NULL` used for C strings. Hence, the entire user name record takes *length*+1 bytes (the +1 is to account for the length of the *length* field).

After the client has received the entire list of the current user names, the client responds with its own user name sent in the same format, i.e., *length string*. Following the sending of this message, the client can assume that it has completed the initial handshake.

At the server side, the only problem that could be encountered at this point is that the new client is using the same user name as an existing one. In this case, the server will simply close the connection to the client and record the event in its log file. The client will interpret this event as a connection error and terminate.

Connection Maintenance

A connected client is expected to send to the server a message at least once every 30 seconds to maintain connectivity. If the server receives no message from a client for 30 seconds, it will close the connection to the corresponding client and consider the respective user as having left the chat.

The client can close the connection explicitly by simply closing its end of the connection (socket). If the client has nothing to send to the server (but still wants to maintain its connection to the server), it should send a dummy *keep-alive* (zero length) message (see below).

Chat Messages

The traffic from a client to the server consists solely of chat messages, which, once received by the server have to be forwarded by the server to **all** the currently connected clients, **including** the client that sent it. The format of these messages is as follows:

messagelength message

where *messagelength* is a 16-bit unsigned integer sent in network byte order indicating the length of the *message* that follows it. The *message* is the message body and is of length exactly *messagelength* bytes.

A keep-alive message is a message of *messagelength* equal to zero (hence not followed by a *message*).

The traffic from the server to each client consists of messages to two types: user-update messages and chat messages. A chat message forwarded by the server to a client looks like this:

`0x00` *length string messagelength message*

where `0x00` is a single byte value (shown here in hexadecimal) followed by the *length*, *string* pair indicating the user name of the user who sent the message, followed by the actual message in *messagelength*, *message*. The same conventions as above apply on how *length*, *string*, *messagelength*, and *message*.

User-Update Messages

User-update messages are only sent by the server to the client(s). A user update indicates that either a new user has joined or an already connected user has exited/terminated/quit. The message formats are, respectively for join and leave:

`0x01` *length string*

and

`0x02` *length string*

In both cases, the initial single byte (shown here in hexadecimal) is followed by a single user name in *length*, *string* format.

The clients keep track of the user-update messages sent from the server, such that the clients are aware, at any point, of the currently connected user names.

Client and Server Interface

The client should be callable as the following command:

`chat379` *hostname portnumber username*

here *hostname* is the Internet name of the host running the server daemon, *portnumber* is the server's port, and *username* is an arbitrarily chosen name that the user wishes to use as their own user name in the chat. Your programs will be tested with user names that consist exclusively of printable non-extended ASCII characters.

It is up to you to provide a minimum human-friendly client user interface. It should allow one to, *at least*, (a) list the current list of user names participating in the chat, (b) report the user-membership updates sent by the server, (c) enter typed-in chat message to be sent to the server (and hence all the connected chat users), (d) allow the user to see the chat messages sent by the chat users, when they arrive (with the name of the user who sent the message preceding the message body), and (e) exit, disconnecting from the server when control-C is hit.

A client must be able to terminate *as soon as possible* when it detects an error in the protocol (closing the socket of the connection), informing the user about what happened.

The server should be callable as the following command:

`server379` *portnumber*

where *portnumber* identifies the INET-domain port on which the server will be awaiting connections from the clients. When started, the server turns itself into a daemon and initiates its service. It only (orderly) terminates when a SIGTERM message is sent to it. When this happens, all clients are disconnected, and the server exits, writing a final line in its log file, indicating `Terminating...`

Any error in the protocol with a specific client should be detected and result in a termination of the connection with the particular client (the rest of the client connections should not be, in principle, influenced). Because the server is running without terminal interaction, it will be making use of a log file with filename:

`server379`*procid*`.log`

where *procid* is the process ID of the server process. (Note there is no space between `server379` and *procid* or between *procid* and the `.log` suffix.)

The server log file should include informative reports of at least, (i) the user-updates, and (ii) errors in the communication to particular clients. Recording the chat messages in the file is not a requirement but implementing it might help you in debugging.

Notice that several things can be happening in parallel, especially at the server: new users can be connecting, while existing ones are disconnecting, chat messages are being forwarded, etc. Make sure the updates received by the clients are consistent, and no clients are ever blocked because of the sluggishness of other clients. Likewise, the server should be responsive, i.e., not defer for later an action that it can perform immediately.

Deliverables

Since all solutions have to follow the same protocol, you are allowed (and encouraged) to test your programs by connecting to servers of other groups or have other group's client connect to your server. Of course, you are not allowed to share code.

The language of implementation is C

In the lab machines, always remember to terminate any server daemon processes you have left running before leaving. Generally, ensure that you have not left any stray processes running on your machine before leaving. Violations of this rule may be penalized with mark deductions.

Your assignment should be submitted as a single compressed archive file. The archive should contain all necessary source files as well as a `Makefile`. Calling `make` without any arguments should generate both the client and the server executables. One item that must accompany your submission is a **brief documentation (can be a single `README.txt` file) of your client user interface**, allowing the TAs to use your client without guesswork. You are expected to deliver good quality, efficient, code. That is, **the quality of your code will be marked**.

Monday, October 3, 2016 (+ typo corrected October 6, 2016)