CMPUT 379 - Assignment #3 (10%)

A Chat Program Using TCP Sockets (first draft)

Due: Monday, November 20, 2017, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you experience in developing client-server programs that utilize TCP sockets for communication over the Internet, and I/O multiplexing to achieve concurrency in data processing.

Details

The chat client-server program specified in Assignment 2 relies on FIFOs (named pipes) for interprocess communication. Thus, its use is limited to users that share the same host computer. In this assignment you are asked to write a C/C++ program, called `a3chat` that extends the functionalities of `a2chat` so that multiple users on the same host, or on different hosts connected by the Internet can chat with each other. The program can be invoked as a server or client using:

```
% a3chat -s portnumber nclient
% a3chat -c portnumber serverAddress
```

where

- `portnumber` specifies the port used by the server to listen to incoming connections. (To avoid port number conflicts when running servers in the lab, use port number $9xxx$ where xxx is the last 3 digits of your student ID number.)
- `serverAddress` is the IP address of the server's host specified either as a symbolic name (e.g., `ui07.cs.ualberta.ca`, or `localhost`), or in dotted-decimal notation (e.g., `127.0.0.1`).
- `nclient` specifies the maximum number of clients that can be connected to the server at any instant. For simplicity, assume that `nclient` ≤ 5.

The Client Loop

On start, the client resolves `serverAddress` (e.g., by calling `gethostbyname()`) to get the server's IP address. The client then prompts the user to enter a command line by displaying a suitable prompt message (e.g., `"a3chat_client:"`).

```
Example: % a3chat -c 9123 ui07.cs.ualberta.ca
         Chat client begins (server 'ui07.cs.ualberta.ca' [129.128.41.37], port 9123)
         a3chat_client:
```

In general, each iteration of the client loop polls the `stdin` for a command line. The client sends each valid command line to the server, waits for the server's response line, and processes the response line.

If the command is `'open username'`, the client attempts to connect to the server. The attempt may fail if the server is not running, or if the server responds with an error message (see the `'open'` command below for more details).

In addition, if the client is in a connected state, the client also polls the specific (full duplex) socket used for communication with the server. If connected, the client also processes the chat lines received from the server. The prompt message should appear after displaying any data to the user.

Detecting crashed clients. To help in detecting a client that has terminated unexpectedly, we also require each client that has a TCP connection to the server to periodically send a special message to the server, called the *keepalive* message (see the details below).

The Server Loop

On start, the server displays a suitable message indicating the values specified on the command line.

```
Example: % a3chat -s 9123 5
         Chat server begins  [port= 9123] [nclient= 5]
```

Each iteration of the server loop polls its listening socket (used to detect incoming connections), the `stdin`, and any possible connected client socket. The server processes each command line, forms a response line (if needed), and sends the response line to the client. Commands that cause the server to forward a chat line to multiple recipients generate a line for each recipient. Note that all communication between clients **go through the server**; there is no direct communication between any two clients. The *exit* command issued from the server's `stdin` terminates the server.

On a successful execution of a client's command by the server, the server sends the client a response line that starts with `"[server]"` followed by some suitable information about the execution of the command (or, just `"[server] done"` if there is no such information). On a failed execution of the command, the server's response line starts with `"[server] Error:"` followed by an explanation of the error.

Detecting crashed clients. The server also keeps track of the periodic keepalive messages received from each TCP connected client. If a connected client stops transmitting such messages for a prescribed consecutive number of periods, the server assumes that the client has terminated unexpectedly. The server should close the TCP connection, logs out the client (if the client has an ongoing chat session), and releases any allocated resources.

The Chat Protocol

Similar to Assignment 2, a chat client utilizes the chat service by issuing a sequence of command lines from the following list. As mentioned above, each command line is typically transmitted to the server, and the client program waits to display the server's response.

1. **open username**: Attempts to open a TCP connection to the server. If successful, send the command to request the server to open a new chatting session using the specified `username` as the user's name.

   For an unconnected client, the command fails to establish a connection if the server is not running. For a connected client, the command fails to open a new session if either (a) the server has reached the client limit specified by `nclient`, (b) the client has a session already going on, or (c) there is another client that uses the specified `username` for chatting. If the command succeeds, the server acknowledges both the connection establishment and registering the user.

   ```
   Example:  % a3chat -c 9123 localhost
             Chat client begins (server 'localhost' [127.0.0.1], port 9123)
             a3chat_client: open ibm
             [server] connected
             [server] User 'ibm' logged in
   ```

2. **who**: Get a list of the logged in users.

   ```
   Example:  a3chat_client: who
             [server]: Current users: ibm, dell
   ```

3. **to user1 user2** $\cdots$: Add the specified users to the list of recipients.

   ```
   Example: a3chat_client: to apple ibm dell unknown
            [server] recipients added: apple, ibm, dell
   ```

   In the above example, there is no client with the name `"unknown"`. The server just ignores such username.

4. $<$ **chat_line**: Send chat_line to all specified recipients.

   ```
   Example: a3chat_client: < Let's start the weekly meeting
            a3chat_client:
            [dell] Let's start the weekly meeting
            a3chat_client:
   ```

   In the example, user `dell` is both a sender and recipient. In response, the server forwards the chat line prefixed with the sender's name in brackets to all recipients.

5. **close**: Close the current chat session **without** terminating the client program. The user may subsequently open a new session with a different username.

   ```
   Example: a3chat_client: close
            [server] done
            a3chat_client:
   ```

   Note: the server is **not** required to notify other clients that a session has been closed.

6. **exit**: Close the current chat session, and terminate the client program. If the `exit` command is received from a client, the server does not generate a response line. Otherwise (if the command is generated from the server's `stdin`), the server terminates.

Keepalive Messages

As mentioned above, *keepalive* messages are sent periodically from each client with a TCP connection to the server. The following parameters defines the contents and periodicity of the messages (with some example values):

```
#define KAL_char     0x6    // A non-printable character (e.g., ACK)
#define KAL_length   5      // Number of KAL_char in one keepalive message
#define KAL_interval 1.5    // Client sends a keepalive message every 1.5 seconds
#define KAL_count    5      // Number of consecutive keepalive messages that needs
                            // to be missed for the server to consider that the client
                            // has terminated unexpectedly
```

The parameters are used as follows:

- A keepalive message is composed of `KAL_length` repetitions of character `KAL_char`.
- A client with a TCP connection to the server is expected to send a keepalive message every `KAL_interval` seconds.
- If the server does not receive `KAL_count` consecutive keepalive messages from the client on time, the server assumes that the client has terminated unexpectedly. If so, the server logs out the user (if the client has an ongoing chat session), closes the socket, and frees other resources that may have been allocated to the client.

Activity Reports

In addition, the server is required to keep track of the last time a logged in client has sent a message to the server, and display an activity report of all logged in clients and their recorded times. Keeping track of the last time a logged in client has interacted with the server (by sending some command to the server) enables the server to identify clients that become idle for prolonged times. As well, the server should report any client that has been disconnected due to the lack of receiving the required number of consecutive keepalive messages on time.

The activity report should be displayed periodically (approximately) every 15 seconds in an automatic way. Note that the server does not use the recorded times for any purpose other than displaying them periodically for information.

```
Example: ...
         activity report:
         'ibm' [sockfd= 4]:Tue Mar  2 21:42:34 2017
         'dell' [sockfd= 5]:Tue Mar  2 21:42:48 2017
         'apple' [sockfd= 6]:Tue Mar  2 21:42:00 2017
         'LG' [sockfd= 7]: loss of keepalive messages detected at ..., connection closed
        ...
```

Note: You may implement this feature using, e.g., UNIX *calendar times* (see, e.g., functions $time()$ and $ctime()$), the $alarm()$ system call, and/or threads.

More Details

1. This is an individual assignment. Do not share code.
2. Although many details about this assignment are given in this description, there are many other design decisions that are left for you to make. In such cases, you should make reasonable design decisions that do not contradict the specifications and do not significantly change the purpose of the assignment. Document such design decisions in your source code, and discuss them in your report. Of course, you may ask questions about this assignment (e.g., in the Discussion Forum) and we may choose to provide more information or provide some clarification. However, the basic requirements of this assignment will not change.
3. When developing and testing your program, **make sure you clean up all processes before you logout of a workstation.** Marks will be deducted for processes left on workstations.

Deliverables

1. All programs should compile and run on the lab machines.
2. Make sure your programs compile and run in a fresh directory.
3. Your work (including a Makefile) should be combined into a single tar archive 'submit.tar'.
   (a) Executing 'make' should produce the required executable file(s).
   (b) Executing 'make clean' should remove all files produced in compilation, and all unused files.
   (c) Executing 'make tar' should produce the 'submit.tar' archive.
   (d) Your code should include suitable internal documentation of the key functions.
4. Typeset a project report (e.g., one to three pages either in HTML or PDF) with the following (minimal set of) sections:
   – **Objectives:** state the project objectives and value from your point of view (which may be different from the one mentioned above)
   – **Design Overview:** highlight in point-form the important features of your design
   – **Project Status:** describe the status of your project; mention difficulties encountered in the implementation
   – **Testing and Results:** comment on how you tested your implementation
   – **Acknowledgments:** acknowledge sources of assistance
5. Upload your tar archive using the **Assignment #3 submission/feedback** link on the course's web page. Late submission (through the above link) is available for 24 hours for a penalty of 10%.
6. It is strongly suggested that you **submit early and submit often**. Only your **last successful submission** will be used for grading.

Marking

Roughly speaking, the breakdown of marks is as follows:

17% : successful compilation of a complete program that is: modular, logically organized, easy to read and understand, and includes error checking after important function calls

03% : ease of managing the project using the makefile

70% : correctness of executing the **a3chat** program

10% : quality of the information provided in the project report

---