The Specification of Project-1

EE122 Introduction to Communication Networks, Fall 2002

Weidong Cui, Karthik Lakshminarayanan, Khian Hao Lim

Revision : 1.23

1 Overview

In this project, you will build a chat service with a client-server architecture in project groups of two. Your job is to write server and client software in C or C++ that fulfils our requirements and handles errors gracefully. Your implementation is required to use TCP sockets for communication. Implementation and testing should be done on the unix instructional machines. Instructional support will be available only for programs written and tested on these machines. The list of chat functionalities you are required to implement is a small subset of the possible ones in a typical Internet Relay Chat service. However, you will be exposed to most of the networking components typically used.

2 Main Components

- **Server**:
  Usage: chatserver <port>
  The single-threaded chatserver listens at a known host and port, listening to new connections from clients, accepting them and handling the commands (refer to section 5) sent from each client. Each chatclient sends a message to the chatserver periodically to inform the server that it is alive. The chatserver can concurrently have up to MAXNUMCLIENTS chatclients using its chatservice (if this state is reached, we call the chatserver “full”). When a chatclient requests to take part in the chatservice run by a chatserver which is already full, its request will be rejected. The chatserver must handle all possible errors and remain alive until explicitly terminated (e.g., by its user typing Ctrl-C).

- **Client**:
  Usage: chatclient <server address> <server port> <username>
  Once started, the single-threaded chatclient connects to the chatserver with a unique username not already used by any clients connected to the chatserver. It then reads user input from standard input and sends commands to the server. It also receives and handles commands from the server. If the chatclient perceives any possible error from the server, it will shut down with an informative message.

3 Implementation Constraints

- **Fixed Header Code**
  A header file, chat.h, a README and a Makefile are provided. Parts of the code in chat.h must not be changed. These parts are indicated by comments in the code. The README is provided with questions that you need to answer for submission. It mostly has to do with the conditions under which you have tested your programs and the names of the instructional machines you tested them on.
• Sample Input and Output

A set of inputs and outputs for a sample session is given for reference. The session demonstrates the exchange between 2 chatclients and a chatserver. It includes five files: chatclient1.in, chatclient1.out, chatclient2.in chatclient2.out, and chatserver.out.

• Defined Numbers

chat.h contains values which are already fixed. They include:

/* the maximum number of clients */
MAXNUMCLIENTS = 255;
/* the maximum length of a username in bytes */
MAXNAMELEN = 255;
/* the period within which chatclient needs to send Command_Keep_Alive’s (in seconds) */
KEEP_ALIVE_PERIOD = 60;

These are to be used in writing your programs. You are not to hardcode the numerical values into your programs. Various byte offsets are also defined for your convenience.

• Printing to Stdout and Stderr

In the following sections, we specify the output your program should print in different circumstances.

– Unless otherwise stated, each line should end with a newline and be flushed. The delimiter between each word of your input and output should be a single space.

– All the output specified in this document is done to stdout. Since we will be using stdout for testing, you must ensure that only the specified output goes there.

– We do not specify the format of output to stderr. However, judicious use of stderr makes for more elegant error handling. Some suggestions for output to stderr are responses to socket connect errors, socket read errors, and invalid packet format.

4 User Input

User input (which we call UserInput) is obtained from standard input. Each UserInput fits into a single line and the chatclient processes the UserInput when the user types <Enter> key. If user input does not conform to any of the following, chatclient prints the following to stdout and the non-conforming UserInput is ignored.

incorrect userinput format

The possible UserInputs are:

1. UserInput_Message

   • Function: Sends a message from this chatclient to other chatclients connected to the chatserver using the names specified in this UserInput. Specifying a username more than once for this UserInput is fine but the chatserver must ensure that the chatclient in question will only receive the message once.

   • Format: /mesg number_of_receivers username_1 username_2 ... username_n message

2. UserInput_List

   • Function: List all the usernames of chatclients connected to the chatserver.

   • Format: /list

3. UserInput_Help

   • Function: List all the commands possible for this chatclient. The chatclient prints a list of possible UserInputs to stdout.
• **Format:** /help
  You can use any format to print the list of commands to stdout. This command is for the sake of completeness of the chatclient.

4. **UserInput_Quit**

  • **Function:** Closes the connection from this chatclient to the chatserver. Chatclient prints the following to stdout and terminates.
  quitting chatclient gracefully
  • **Format:** /quit

5 **Protocol**

A *command* is the basic unit of data transfer between a server and each client. There are four distinct packet formats for this protocol. The formats along with the size of each field are illustrated in Figure 1, Figure 2, Figure 3 and Figure 4.

The **Type** field (first 2 bytes) identifies the command. The **Length** field refers to the length of the application layer portion of the packet (i.e., not including the TCP/IP portion), and the **Num** field refers to the number of usernames that follow in the List of Users. Each **Username** field in the packet format does not include the '\0' ending and the corresponding **Length** field counts the number of bytes for the **Username** excluding the '\0' at the end. The chatclient also removes the newline (that it reads from stdin) while filling the message field that it includes with a Command_Message.

The numbers in the packet are in network byte order (refer to htonl(3), htons(3), ntohl(3) and ntohs(3)).

The following are the list of commands as identified by the **Type** field.

---

1 The number in parenthesis is the Unix man page on which documentation of the command resides.
1. Command_Join:

- **Packet Format:** Use the Join packet format (Figure 2).
- **Category:** client-to-server
- **Sender action:** This is the first command that the chatclient sends to the chatserver and is a request to join the chat service. The requested username is also sent in this command.
- **Receiver action:** When the chatserver receives this packet, one of the following situations could arise:
  (a) The chatserver allows the chatclient to join and does not send any response. Chatserver prints the following to stdout:
      join: <source_user_name>
  (b) The name is already in use by one of the other chatclients who joined the chatserver, then the chatserver sends a *Command_Error_Name_Used* command. Chatserver prints the following to stdout:
      disconnected: <user_name> existing user name
  (c) The server is already full. If the server already has MAXNUMCLIENTS connected to it, it should accept the new connection, send back a *Command_Error_Server_Full* to the client, and close the connection. The chatserver prints the following to stdout:
      disconnected: server full
  (d) An erroneous scenario occurs when the first command that a chatclient sends to the chatserver is not *Command_Join*. This is treated as a command type that the chatserver does not recognize and is handled by *Command_Error_Unknown_Command*. The actions performed are explained in *Command_Error_Unknown_Command* section.

2. Command_List_Request:

- **Packet Format:** Use the Basic packet format (Figure 1).
- **Category:** client-to-server
- **Sender action:** Upon receiving *UserInput_List*, chatclient sends this to the server.
- **Receiver action:** Upon receipt, the chatserver responds by sending back a *Command_List_Reply* containing the list of usernames of chatclients connected to the chatserver. The chatserver also prints the following to stdout:
      list_request: <source_user_name>

3. Command_List_Reply:

- **Packet Format:** Use the List packet format (Figure 3).
- **Category:** server-to-client
- **Sender action:** Upon receiving *Command_List_Request*, the chatserver responds by sending this command back, which contains a list of user names of all clients that have connected to the chatserver, including the one that sent the *Command_List_Request*.
- **Receiver action:** The chatclient prints out the list of users in to stdout as follows:
      list: <user_name_1> <user_name_2> ... <user_name_k>

4. Command_Message:

- **Packet Format:** Use the Message packet format (Figure 4).
- **Category:** client-to-server
- **Sender action:** When the user types *UserInput_Message*, the chatclient sends this command. Usernames that are repeated on the *UserInput_Message* may be forwarded as well.
5. Command_Message_Forward:

- **Receiver action:** The chatserver responds by forwarding the message using Command_Message_Forward to all the clients with usernames listed in this command. The chatserver ensures that the chatclient whose username is repeated gets Command_Message_Forward only once. The chatserver prints the following to stdout:
  
  \[
  \text{msg: <source_user_name>}
  \]

  For each username that does not correspond to a chatclient, chatserver prints the following to stdout:
  
  \[
  \text{msg: <source_user_name> to nonexistent user <non_existent_user>}
  \]

6. Command_Keep_Alive:

- **Packet Format:** Use the Message packet format (Figure 4). There should be exactly one username specified in the List of Users in the packet. This is the username of the sender.
- **Category:** server-to-client
- **Sender action:** Upon receiving Command_Message, the chatserver sends this command, containing the message, to each of the clients whose usernames are listed in the Command_Message command. Any username which does not correspond to a user is ignored. Any client whose username appears in the list will receive the message once regardless of the number of times his username appears in the Command_Message command.
- **Receiver action:** The chatclient prints the following to stdout:
  
  \[
  \text{msg: <source_user_name>: <message text>}
  \]

7. Command_Error_Unknown_Command:

- **Packet Format:** Use the Basic packet format (Figure 1).
- **Category:** server-to-client
- **Sender action:** Upon receiving a command type that the chatserver does not recognize, or a format that does not conform to the packet format stipulated, or a command from a chatclient who has not joined yet (refer to Command_Join), the chatserver sends this command to the client, closes the connection and drops any state held for that client. Chatserver prints the following to stdout:
  
  \[
  \text{disconnected: <user_name> sent unknown command}
  \]

- **Receiver action:** Upon receiving this, chatclient prints out an error, closes the connection to the server, and shuts down. The chatclient prints the following to stdout:
  
  \[
  \text{disconnected: server received unknown command}
  \]

8. Command_Error_Server_Full:

- **Packet Format:** Use the Basic packet format (Figure 1).
• **Category:** server-to-client
• **Sender action:** When there are already MAXNUMCLIENTS connected to the chatserver and it receives a new connection, it sends this command and closes the connection.
• **Receiver action:** Upon receiving this command, the chatclient prints out a message, closes the connection, and shuts down. Chatclient prints the following to `stdout`:
  disconnected: server full

9. **Command_Error_Name_Used:**
• **Packet Format:** Use the Basic packet format (Figure 1).
• **Category:** server-to-client
• **Sender action:** Upon receiving a `Command_Join` from a chatclient that gives a username that is in use by one of the chatclients already connected to the server, the chatserver sends this command back. The chatserver then closes the connection and removes any state for this chatclient.
• **Receiver action:** Upon receiving this command, the chatclient prints out a fixed message, closes the connection and shuts down. Chatclient prints the following to `stdout`:
  disconnected: name in use

6 Other Protocol and Printing Issues

• **Ending Chatservice**
  The chatclient quits after receiving `UserInput_Quiit`. The chatclient simply closes the TCP socket to the chatserver and exits. The corresponding socket on the chatserver side will read 0 bytes and understand that the chatclient has left the chatserver. Under this condition and other socket errors, the chatserver prints the following to `stdout`:
  disconnected: <user_name> closed connection
  Under this condition and other socket errors, the chatserver closes the TCP socket and removes the state for that particular chatclient.

7 Implementation Suggestions

• **Standardized Types and Definitions**
  `chat.h` contains typedefs and defines for some standardized types often used in socket programming. It is useful to stick with these typedefs. Of course, you are free to use your own definitions.

• **Execution Model**
  Because the programs are single threaded, they must not block on reads or writes for sockets or standard input. I/O should be non-blocking. Socket and file descriptor activity (standard input and output streams are controlled by file descriptors as well) should be detected through the `select(2)` system call. Chapter 6 of Unix Network Programming by W. Richard Stevens addresses many of the issues you will need to understand.

• **Socket Programming API**
  Some functions you probably need to know well include: `open(2)`, `close(2)`, `connect(2)`, `setsockopt(2)`, `fcntl(2)`, `connect(2)`, `listen(2)`, `select(2)`, `socket(2)`, `accept(2)`, `bind(2)`, `htons(3)`, `htonl(3)`, `ntohs(3)`, `ntohl(3)`, `gethostbyname(3)`, `gettimeofday(2)`, `perror(3)`.

• **Start Early**
  This cannot be stressed enough.
8 Checkpoint

In order to monitor your progress, we decided to have a “checkpoint version” submitted by Oct 2 before 11.59 pm. Subsequently, there will be a meeting with a TA who will check on the progress of your project. Your chatserver and chatclient must demonstrate part of the functionality of the final version, namely it has to handle the following functionalities:

- \texttt{UserInput\_Quit}
- \texttt{UserInput\_Help}
- \texttt{Command\_Keep\_Alive}
- \texttt{Command\_Join}
- \texttt{Command\_Error\_Server\_Full}
- \texttt{Command\_Error\_Name\_Used}

In short, for the checkpoint your chatserver must be able to demonstrate the ability to handle normal and error conditions for new connections. And your chatclient must be able to demonstrate the ability to connect to a chatserver and remain alive. It must also quit gracefully after receiving \texttt{UserInput\_Quit} or other errors like \texttt{Command\_Error\_Name\_Used} or \texttt{Command\_Error\_Server\_Full}.

9 Testing

You must ensure that your chatserver and chatclient implementation meets the following requirements.

- Consistent with our specification: Your chatclient and chatserver must be able to work with other chatclients and chatservers that implement the same set of protocols. Your implementations will be cross tested with other implementations for compatibility.
- Handles errors gracefully: Your programs should do their best to continue function correctly despite non-specified behavior. For example, regardless of incorrect input, your chatserver should never crash or hang. Your implementations will be tested with erroneous input to ensure graceful handling of errors.

10 Submission

Project submission will be done as in other classes. To submit the checkpoint, use the following command:

\begin{verbatim}
% submit proj1-checkpoint
\end{verbatim}

To submit the final version of your project, use the following command:

\begin{verbatim}
% submit proj1
\end{verbatim}

The class website and newsgroup will be the official authority for information on submission or due dates. The files required for submission include the files that we have provided you. The questions in README should be answered as well.