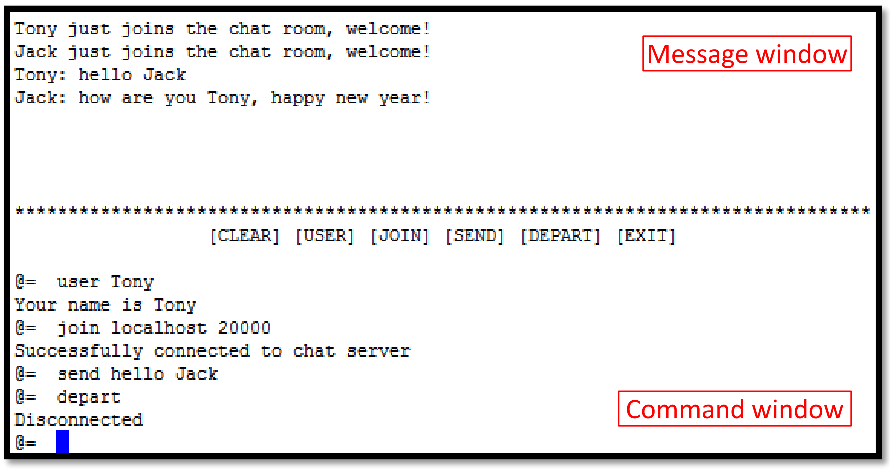
CSIS0234_COMP3234 Computer and Communication Networks
Programming Project - Simple Internet Chat Program

Release Date: Feb. 24, 2014 (Monday)
Due Date: 5:00pm, Apr. 24, 2014 (Thursday)

1. Overview

The client/server application chosen is a simple internet chat program (SICP) on the Linux platforms. You are going to write ONE program only, *the server program*, and the client chat program is available for you. The client process communicates with the server process by establishing a TCP connection. As the SICP involves more than two processes, which are communicating via message passing, we need to adopt a standard *chat protocol*. This allows the standard client program to talk to your newly implemented server program.



```
Tony just joins the chat room, welcome!
Jack just joins the chat room, welcome!
Tony: hello Jack
Jack: how are you Tony, happy new year!




*******************************************************************************
                  [CLEAR] [USER] [JOIN] [SEND] [DEPART] [EXIT]

@=  user Tony
Your name is Tony
@=  join localhost 20000
Successfully connected to chat server
@=  send hello Jack
@=  depart
Disconnected
@=
```

Figure 1: Typical usage/output of a chat client in Linux terminal.

2. Design

The chat server is designed to have only one chat room. The server adopts multiple threads to concurrently serve multiple chat clients, so that the main thread can continue to accept new clients. Each *client_thread* is responsible to receive messages from the corresponding client process, and puts the messages in a message queue – *MsgQue*. The server uses a dedicated *broadcast_thread* to send the messages in MsgQue to all clients, one by one.

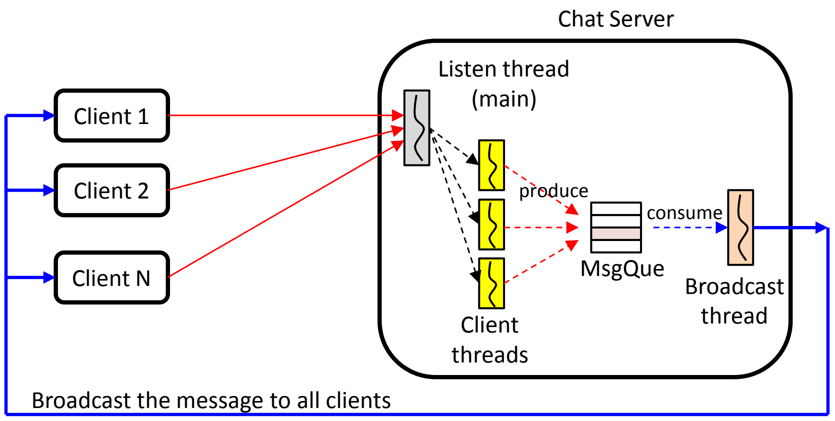


Figure 2: Conceptual architecture of chat client/server program.

2.1 Client-side Program Specification

The client program is the user interface in which it accepts input(s) from the user. The user interface of the client program appears as follows. The client interface makes use of the *ncurses* library to manipulate the data input/output to the terminal. You are NOT required to write the chat client program - ***chat_client***.c, as we have provided it to you. Once the program started, it accepts commands/instructions from the user. There are total 6 commands available for the user, and their actions are specified as follows.

- **USER/user [username]** – the argument *username* is used for setting the user identity of all messages posted to the chat server. As the chat server needs to display all messages with their associated usernames, the client program MUST get the user's username before setting up the connection to the chat server. A user can rename his/her *username* by using this command (if current username clashes with other user's username).
- **JOIN/join [server name] [port number]** – to connect to a chat server. A TCP connection will be established to the targeted chat server, and then a JOIN message (with user's username) will be sent to the server. If the chat client has already connected to a server, it is not allowed to join another chat server before termination of current TCP connection.
- **SEND/send [message]** – to send a message to the chat server. The message will be sent to the server via the established TCP connection, and will be shown in every client's message window (broadcasted by the server). The client must be connected to the server before sending a message.
- **DEPART/depart** – to terminate current chat connection. The user does not exit from the program, and is able to rejoin the server again.
- **EXIT/exit** – the user exits from the program.
- **CLEAR/clear** – the command window will be cleared.

The state diagram in Figure 3 shows the basic idea of how these commands are used.

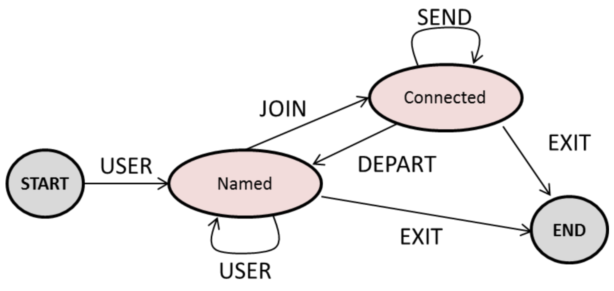


Figure 3: The state diagram of the client-side program.

2.2 Server-side Program Specification

The behavior of your *chat_server* program MUST match with the behavior of the provided client program. Without this compatibility, the *chat_client* program cannot talk with your server program successfully. Below are the flowcharts of the *chat_server* program, which you can use as the starting point to develop your server program.

The server program accepts one optional input argument – *listen_port_number*. If this argument is missed, the server will use the default port number 3500 (defined in the provided header file *chat_server.h*). Once it starts, it runs forever. To terminate the server program, user can use SIGINT (kill -2 or Ctrl + C) or SIGTERM (kill -15) signals to terminate the chat server and all its threads.

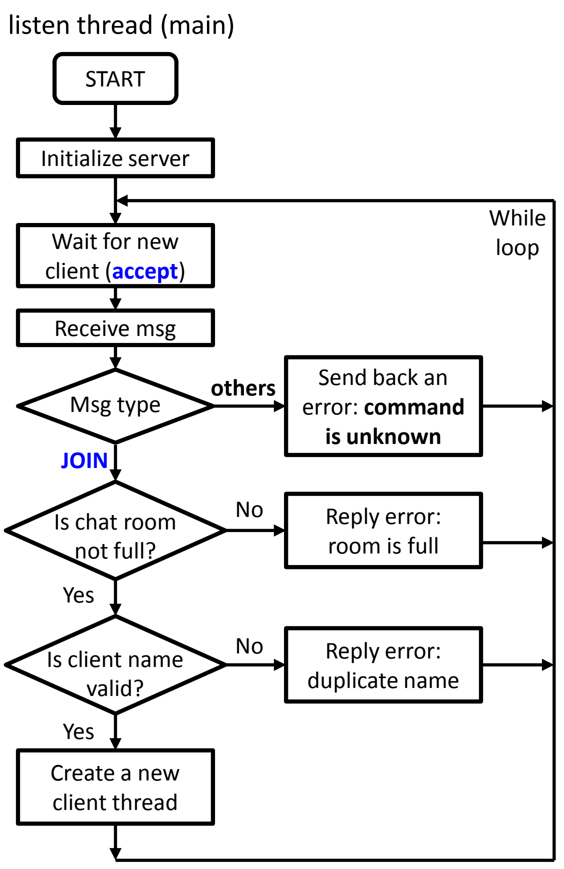

listen thread (main)
START
Initialize server
Wait for new client (accept)
Receive msg
Msg type
others
Send back an error: command is unknown
JOIN
Is chat room not full?
No
Reply error: room is full
Yes
Is client name valid?
No
Reply error: duplicate name
Yes
Create a new client thread
While loop

Figure 4: The flowchart of the server's *main thread*, to listen for new connections.

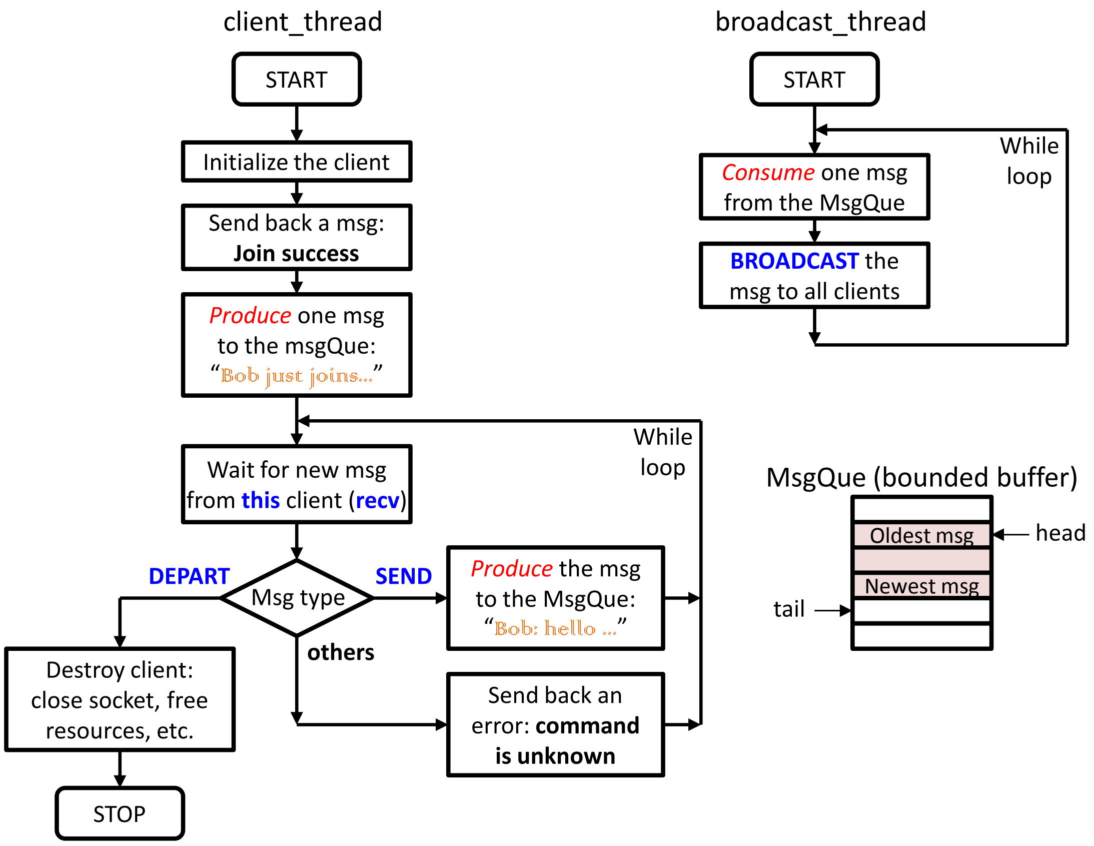

client_thread
START
Initialize the client
Send back a msg: Join success
Produce one msg to the msgQue: "Bob just joins..."
Wait for new msg from this client (recv)
While loop
Msg type
DEPART
SEND
others
Produce the msg to the MsgQue: "Bob: hello ..."
Send back an error: command is unknown
Destroy client: close socket, free resources, etc.
STOP
broadcast_thread
START
Consume one msg from the MsgQue
BROADCAST the msg to all clients
While loop
MsgQue (bounded buffer)
Oldest msg
head
Newest msg
tail

Figure 5: The flowcharts of *client_thread* and *broadcast_thread*, both manipulating MsgQue.

2.3 Chat Protocol

The following tables describe the standard chat protocol designed for this assignment. To make life easier, you are provided with a header file - ***chat.h***, which contains the required message structure used by this client/server communication. In order to make sure that the standard *chat_client* program could communicate with your *chat_server* program, you are NOT allowed to modify, add and change any definition appeared in ***chat.h***, as amendment made to this file may result in incompatibility.

Table I: The format of "message" exchanged between clients and the server (see *chat.h*).

Field: **instruction (int)**	Field: **private data (int)**	Field: **message content (string)**

Table II: Message types sent by the *chat_client* program (see *chat.h*).

Instruction	**Private data**	**Message Content**
CMD_CLIENT_JOIN	Length of the client name	Client name
CMD_CLIENT_SEND	Length of the message	Chat message
CMD_CLIENT_DEPART	-1	Nil

Table III: Messages types sent by the *chat_server* program (see *chat.h*).

Instruction	**Private data**	**Message Content**
CMD_SERVER_JOIN_OK	-1	Nil
CMD_SERVER_BROADCAST	Length of the message	Broadcasted message
CMD_SERVER_CLOSE	-1	Nil
CMD_SERVER_FAIL	ERR_JOIN_DUP_NAME ERR_JOIN_ROOM_FULL ERR_UNKNOWN_CMD	Nil

The following is a detailed specification on the interactions of the chat protocol: how the client(s) talk to the server, and how the server responses to different requests/instructions from multiple clients.

- When a client issues CMD_CLIENT_JOIN instruction, if successfully joined, the server should return CMD_SERVER_JOIN_OK to this client, and should put a welcome message "<client name> just joins the chat room, welcome!" in the MsgQue. Otherwise, CMD_SERVER_FAIL should be returned with the specific error code, e.g. ERR_JOIN_DUP_NAME (duplicate name), ERR_JOIN_ROOM_FULL (room is full), etc.
- When receiving the client CMD_CLIENT_SEND instruction, the server should put the message (add the username as prefix) in the MsgQue. After that the *broadcast_thread* should automatically send the message to all clients, using CMD_SERVER_BROADCAST instruction.
- When receiving the client CMD_CLIENT_DEPART instruction, the server should notify all clients (by broadcast) about this departure, and terminates the corresponding *client_thread*. It is OK to not send the departure message to the departing client.
- When the server received a termination signal (e.g., Cltr-C), it should notify all clients with CMD_SERVER_CLOSE instruction, the clients should exit from the program after receiving this instruction.
- When a client issues an unknown command, the server should return CMD_SERVER_FAIL, with error code being ERR_UNKNOWN_CMD.

3 Program

3.1. Source code

You will be provided with the following files:

(1) The standard client-side program ***chat_client.c***
(2) The *undone* server-side program ***chat_server.c*** -- **for you to finish.**
(3) The header file ***chat_server.h***, used by *chat_server.c*
(4) The header file ***chat.h***, used by both *chat_client.c* and *chat_server.c*
(5) The ***Makefile***, for compilation under *Linux* platform.

3.2. Run the program

You are recommended to develop and run your program under *Linux*. The provided sample code has been tested under Ubuntu 13 (installed in HW 311/312 Labs) and CentOS 6. It should also work on other Linux distributions. In case you find incompatibility problem when compiling the source code in your own PC, please inform us.
If you would like to develop your program under SunOS platform, e.g. using CS servers: {belief, honest, virtue, genius}.cs.hku.hk, please inform the tutor to get another Makefile.

To compile the program, please use *make* under Linux terminal:

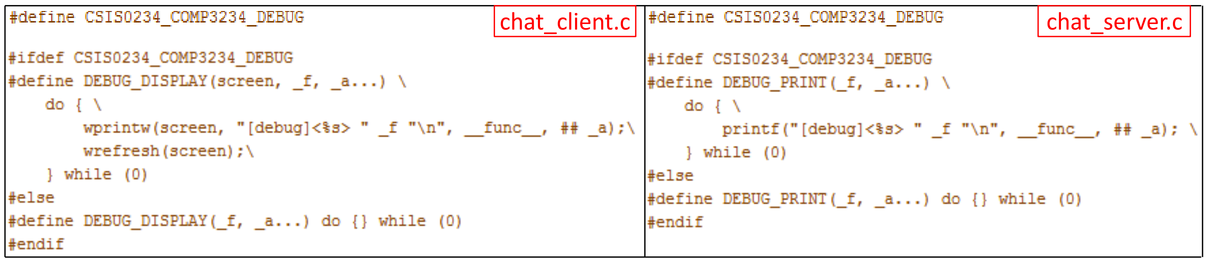
```
[username@hostname] make chat_client  # compile the standard client program
[username@hostname] make chat_server  # compile your server program
[username@hostname] make clean        # clean all
```

To run the program:

```
[username@hostname] ./chat_server <listen_port_number>
[username@hostname] ./chat_client
```

3.3 Debug your program

To help you avoid writing redundant code for debugging, the following macros have been prepared for you, which are very easy to use. Please refer to *chat_client.c* to see how to use `DEBUG_DISPLAY()`. The usage of `DEBUG_PRINT()` is the same as `printf()`. When you do not need verbose messages, just comment out `CSIS0234_COMP3234_DEBUG` in your code.



chat_client.c

```
#define CSIS0234_COMP3234_DEBUG

#ifdef CSIS0234_COMP3234_DEBUG
#define DEBUG_DISPLAY(screen, _f, _a...) \
    do { \
        wprintw(screen, "[debug]<%s> " _f "\n", __func__, ## _a);\
        wrefresh(screen);\
    } while (0)
#else
#define DEBUG_DISPLAY(_f, _a...) do {} while (0)
#endif
```

chat_server.c

```
#define CSIS0234_COMP3234_DEBUG

#ifdef CSIS0234_COMP3234_DEBUG
#define DEBUG_PRINT(_f, _a...) \
    do { \
        printf("[debug]<%s> " _f "\n", __func__, ## _a); \
    } while (0)
#else
#define DEBUG_PRINT(_f, _a...) do {} while (0)
#endif
```

4. Submission & Marking Scheme

In order to help you *incrementally* finish this programming assignment, we set up this *multi-stage* checking/marking scheme, with each stage having a very clear learning objective. Under this scheme, it is *not easy* (hopefully) for you to lose all marks, and meanwhile, you indeed need to pay certain effort to get full marks.

[Stage 1: 25%]

In this stage, your (single thread) server program only needs to interact with **one** client.
Objective: To learn basic socket programming skills. Even if you don't have the knowledge in multi-threaded programming, it is not difficult to finish this stage.

Test cases: When the client issues JOIN instruction to the server, the server should return the message of "<client name> just joins the chat room, welcome!"; when the client issues SEND instruction to the server, the server should send the message back to the client to display. After that, both server and client can exit.
Submission [Optional]: Your program should be included in a *self-contained* archived file, and all source code files should be there (*.c, *.h and Makefile).
Please name the archived file as: *c0234_uid_username_stage1.tar* (or .zip).
Deadline: 9:00am, Mar. 06, 2014 (Thursday, just before Lab 2).

[Stage 2: 25%]
In this stage, when **multiple** clients join the chat server, your server program must create one thread for each newly joined client. To make it simple, the server just echoes back whatever one client sends to it (no need to broadcast to other connected clients).
Objective: To learn and use multi-threading techniques to handle simultaneous connections. Generally, this is how web server applications process user requests in the real world.
Test cases: Similar to that in stage 1, but the server should be able to simultaneously accept instructions from multiple clients as well as respond to error situations.
Submission [Optional]: Your program should be included in a *self-contained* archived file, and all source code files should be there (*.c, *.h and Makefile).
Please name the archived file as: *c0234_uid_username_stage2.tar* (or .zip).
Deadline: 9:00am, Mar. 20, 2014 (Thursday, just before Lab 3).

[Stage 3: 35%]
In this stage, you are going to implement the **SEND and DEPART** functionalities. Basically, the *client_threads* place messages to the MsgQue, and the *broadcast_thread* consumes messages (see Figure 2) from the MsgQue. You should use a bounded-buffer to implement this producer-consumer interaction.
Objective: To understand how a multi-threaded server handles inter-thread communication and coordination.
Test cases: When one client issues SEND instruction, the server should *broadcast* this message (add the username as prefix) to all connected clients; when one client issues JOIN/DEPART instruction, the server should inform the others with the message "<client name> just joins the chat room, welcome!" or "<client name> just leaves the chat room, goodbye!". For the goodbye message, it is OK that the departing client does not receive it.
Submission [Optional]: Your program should be included in a *self-contained* archived file, and all source code files should be there (*.c, *.h and Makefile).
Please name the archived file as: *c0234_uid_username_stage3.tar* (or .zip).
Deadline: 9:00am, Apr. 03, 2014 (Thursday).

[Stage 4: 15%]
In the final stage, you are going to implement **SERVER SHUTDOWN** functionality.
Objective: To handle proper server program shutdown, and cultivate good programming habits.
Test cases: When the chat server is terminated (e.g. <Ctrl + c>): (1) all clients should be notified so that they can react properly. (2) All dynamically allocated resources should be released, and all opened sockets should be closed. The tutor will check your source code.
Final Submission [Compulsory]: Your program should be included in a *self-contained* archived file, and all source code files should be there (*.c, *.h and Makefile).
Please name the archived file as: *c0234_uid_username_final.tar* (or .zip).
Deadline: 5:00pm, Apr. 24, 2014 (Thursday).

Grading Policies:

1. The tutor will first test your *final submission* after the project deadline. If your program fully complies with the project specification, you'll get all the marks for all four stages automatically. Otherwise, tutor will test your *Stage_3 submission* (if any). If it passes all test cases, you'll get all the marks for the first three stages. Otherwise, tutor will test your *Stage_2 submission* (if any). If it passes all test cases, you'll get all the marks for the first two stages. Otherwise, tutor will test your *Stage_1 submission* (if any) to give marks.
2. If you want to get those stage marks *earlier*, you are allowed to demonstrate your program to the tutor, during the consultation hours or by appointment. This is recommended for those who have difficulties in finishing all stages (e.g. when you are in a rush for multiple deadlines).
3. You do not need to care about the format of display messages. Your program is marked according to whether the required ***functionalities*** have been correctly implemented.
4. If you have modified some part of the *chat_client.c* program which leads to very different inputs/outputs, you must submit a short report to describe how your program fits the above marking scheme. Otherwise, you do not need to submit this report.
5. Your submission(s) will be primarily tested under Ubuntu 13 (installed in HW311/312). However, when testing cross-architecture capability, we may involve genius.cs.hku.hk to test running the program. Make sure that your program can be compiled *without any error or warning*. Otherwise, we have no way to test your submission(s) and thus you have to lose *all* the marks.
6. As the tutor will check your source code, please write your program with good readability (i.e., with good code convention and sufficient comments) so that you will not lose marks due to possible confusions.

-END-