

Assignment - Internet Chatting

Hoang Trinh (hoang@cs.uchicago.edu)
Wonseok Chae (wchae@cs.uchicago.edu)

Document History

- April 22, 2005 - Original version
- April 26, 2005 – Message formats were changed to make implementation easy.
- April 26, 2005 – Add “Add_User” / “Remove_User” messages.
- May 9, 2005 – Synchronize this document with the real implementation.
- May 11, 2005 – Add test cases.
- May 19, 2005 – Add some comments on “Reliability”

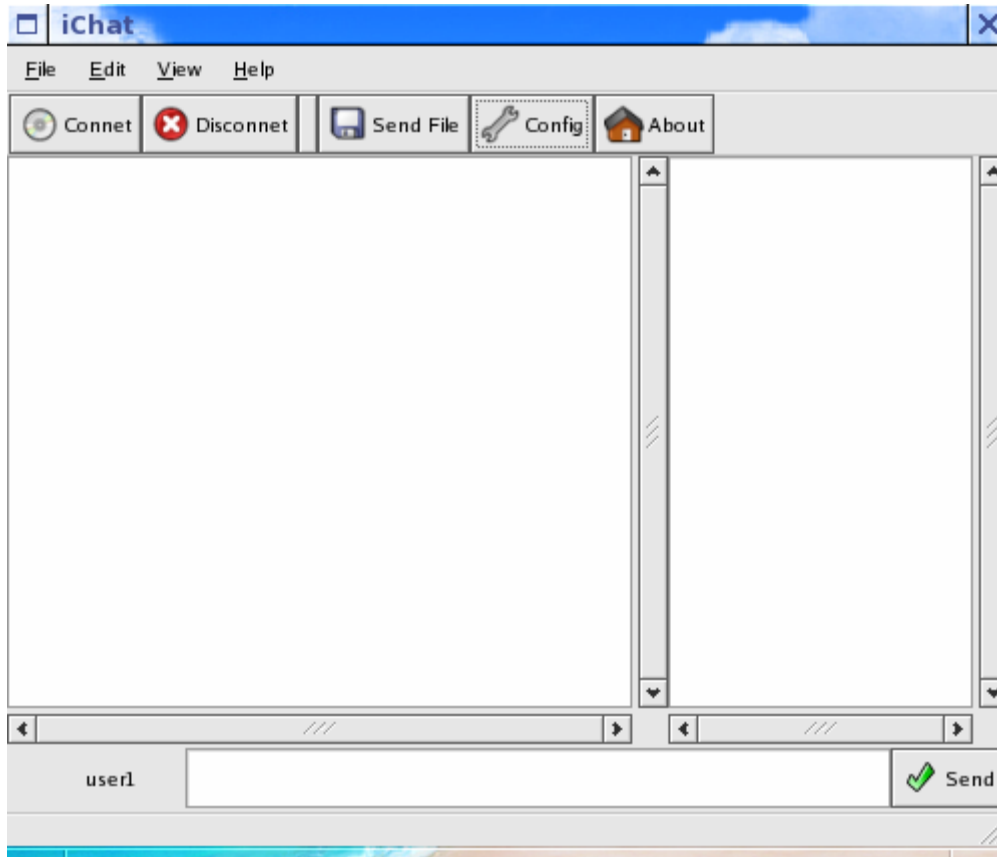
Requirement Analysis

(It was based on the assignment handout from Prof. Beazley.)

- Support Internet chatting.
- Consist of two components: the chat client and the chat server
- The chat server keeps track of users and makes it possible for users to find each other online.
 - The chat server is only used to keep track of active clients.
 - When a client starts, it contacts the chat server and registers itself.
 - When a client wants to chat with another client, it contacts the server to locate the machine and port number of the client.
 - Connections between the chat client and the server may use TCP.
- The chat client runs by users for chatting with each other.
 - Communication occurs directly between clients.
 - Any number of uses must be able to chat with each other.
 - The chat client must be able to send simple text. It must be done with UDP. Furthermore, the client should only use a single UDP socket for all communication.
 - The chat client must be able to send a file that is specified by the user. When files are sent between clients, TCP may be used.
 - The client program must provide some kind of reasonable user interface.
- Both client and server program must be able to handle situations where clients terminate and restart.
- To compile source codes, a makefile should be provided.
- Include a README file containing names and other details.

Client / GUI

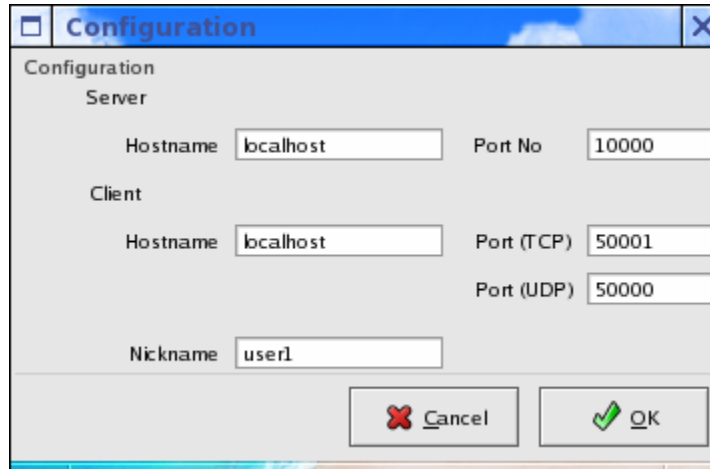
(It was built with Glade-2.0. I am not sure, but I guess the users might need gtk+ in their machine. We need to investigate it is necessary or not.)



- “Connect” button – connect to the chat server and register itself
- “Disconnect” button – disconnect to the chat server
- “Send File” button – send the specified files to the specified users
- “Config” button – set the configurations such as a nickname, host id, port number, etc.
- “About” button – display the information about this program
- “Send” button – send text messages to the all current users

How to run it

- Server
 - \$ chatsrvr portnum
- Client
 - \$ ichtat
 - After running it, you are suppose to set the configuration by pressing “config” button.

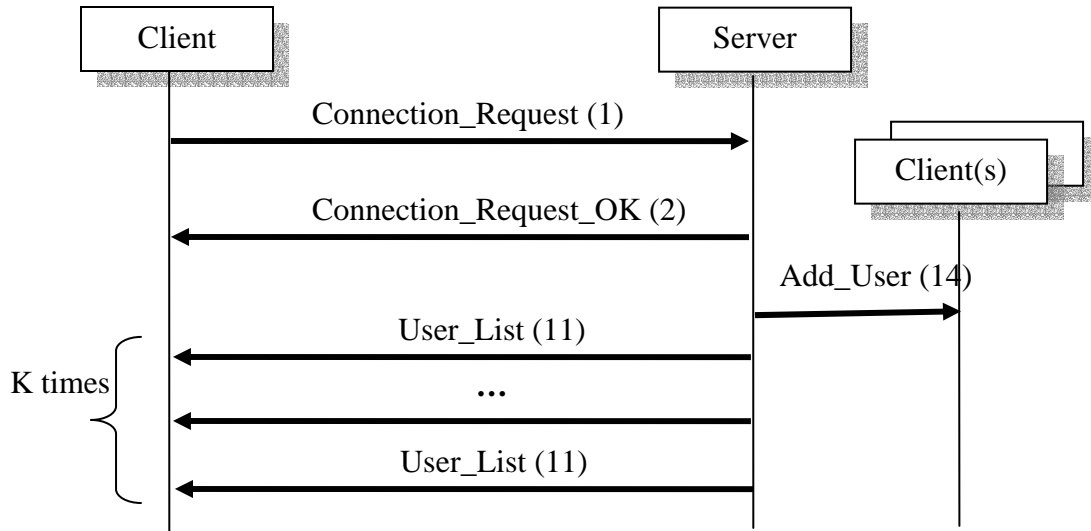


The image shows a Windows-style dialog box titled "Configuration". The dialog is divided into sections for "Server" and "Client" settings. The "Server" section has fields for "Hostname" (localhost) and "Port No" (10000). The "Client" section has fields for "Hostname" (localhost), "Port (TCP)" (50001), and "Port (UDP)" (50000). There is also a "Nickname" field with the value "user1". At the bottom right, there are "Cancel" and "OK" buttons.

Section	Field	Value
Server	Hostname	localhost
	Port No	10000
Client	Hostname	localhost
	Port (TCP)	50001
	Port (UDP)	50000
Global	Nickname	user1

Scenarios of the communications

- Connection / Registration / Update



1. When a user presses the “Connect” button, the chat client program sends a “Connection_Request” message (Type 1) to the chat server.

1 Byte	20	4	2	Comments
Type = 1	Nick Name	Addr	Port #	Addr is not used in the server side. Server can calculate the exact addr with IP header.

2. When the chat server receives a “Connection_Request” message (Type 1) from the client, it returns “Connection_Request_OK” message (Type 2).

1 Byte	1	1	Comments
Type = 2	ID#	K = # of clients	ID# = this client’s ID

Then, send “User_List” message (Type 11) including individual user information per a message. The # of this message should be same as K specified by the previous “Connection_Request_OK” message.

1 Byte	3	1	25	4	2	Comments
Type = 11	unused	ID#	Nick Name	Addr	Port #	

- At the same time, the chat server send “Add_User” message (Type 14) to the existing clients to let them know this incoming client.

1 Byte	1	25	4	2	Comments
Type = 14	ID#	Nick Name	Addr	Port #	

- If there exists the same nick name among the existing users, the server will return “Connection_Request_FAIL” message (Type 16) instead of “Connection_Request_OK”.

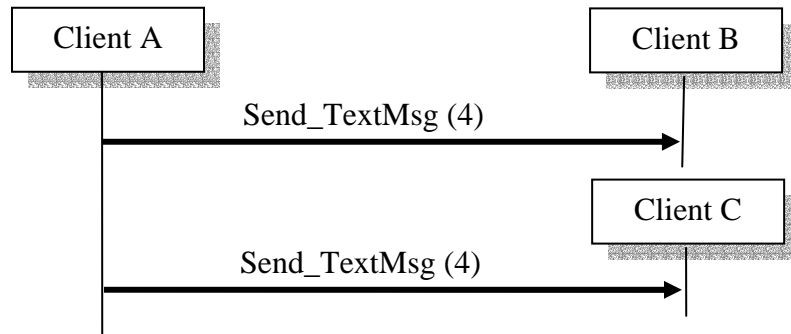
1 Byte	1			Comments
Type = 16	Status = 1			

Similarly, if there are more requests than the chat server can handle at the same time, the server will return “Connection_Request_REJECT” message (Type 15) instead of “Connection_Request_OK”.

1 Byte	1			Comments
Type = 15	Status = 1			

- Upon receiving “User_List” messages, the chat client shows the user list.

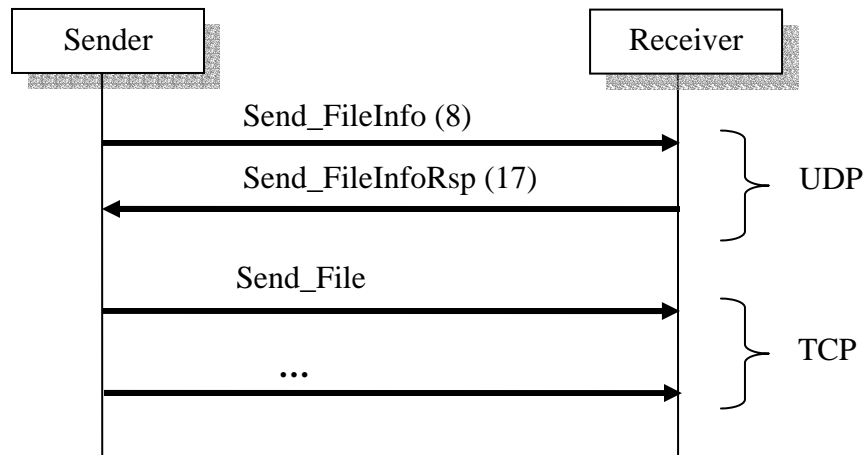
- Text messages sending



1. When a user presses the “Send” button or finishes input by entering “Enter”, the chat client program sends a “Send_TextMsg” message (Type = 4) to all other clients.

1 Byte	1	1	256	Comments
Type = 4	ID#	Len	TextMsg	

- File sending



1. When a user chooses targets among the user list and presses the “Send File” button, the “file selection dialog” will pop up. Once a user specifies which file will be sent, the chat client program sends “Send_FileInfo” message (Type 8) to the specified client to inform the specified file will be sent.

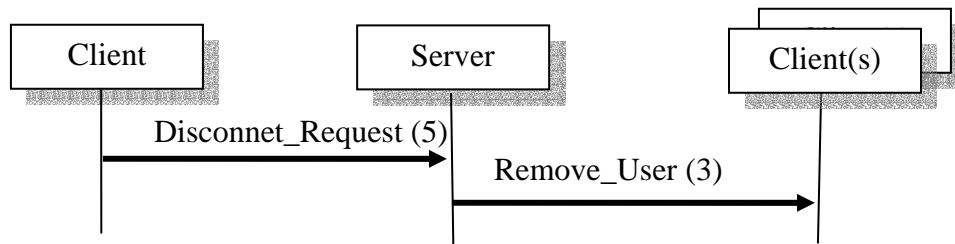
1 Byte	1	4	Comments
Type = 8	ID#	File_Size	Mode tells how to set mode when creating this file
???	100		
mode	File Name		

2. Then, the receiver responses with “Send_FileInfoRsp” message (Type 16) that contains TCP port number and status showing if he/she wants to receive this file or not.

1 Byte	1	1	2	Comments
Type = 17	ID#	status	Port #	Staus = 1 means accept, otherwise the client does want to receive it.

3. Once the sender receives “Send_FileInfoRsp” message with satus=1, it sends files via TCP socket and the receiver starts receiving. If there exists no error, then popup dialog will show saying “successfully sent (or received).”

- Disconnection / Update



1. When a user presses the “Disconnect” button to finish chatting, the chat client program sends a “Disconnect_Request” message (Type = 5) to the chat server.

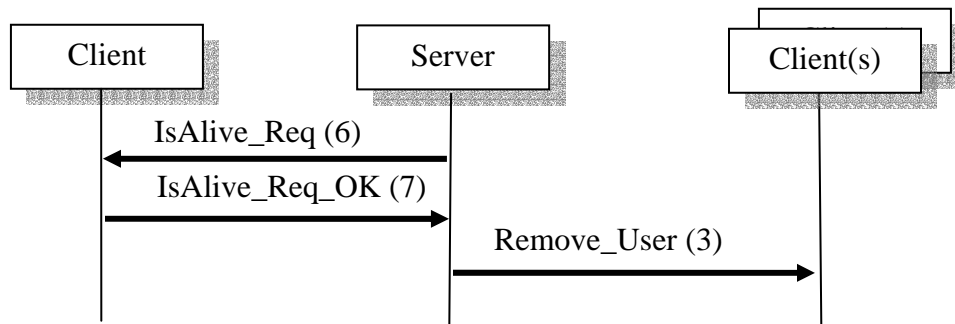
1 Byte	1			Comments
Type = 5	ID#			

2. Once the chat server verifies this request is valid (i.e., the server can match ID# and the IP address (or Port #)), it sends “Remove_User” message (Type 3) to inform that the specified user is going out.

1 Byte	1		Comments
Type = 3	ID#		ID# = this client’s ID Same as Type 2

3. Then, a chat client updates the user list.

- “Client Status” Checking / Update



1. Periodically, the chat server will send a “IsAlive_Req” message (Type 6) to all clients to check whether a client is available.

1 Byte	1			Comments
Type = 6	ID#			ID# = receiver's ID

2. When a client receives a “IsAlive_Req” message from the chat server, it should return “IsAlive_Req_OK” message (Type 7) to inform that it is available (i.e., on line).

1 Byte	1			Comments
Type = 7	ID#			ID# = sender's ID

3. If there is no response from a client, the chat server may conclude that that client is not available and inform other clients with “Remove_User” message (Type 3) as in the same way in the “Disconnection / Update” scenario.

Test Case

- Chat Sever : champion.cs.uchicago.edu
- Client
 - o champion (user1)
 - o tippecanoe (user2) – a little slow
 - o champion (user3)
 - o boardwalk-flyer (user4)

- Test cases

#	Description	Results	Comments
1	Run server (%chatserver 10000 &) on champion.cs.uchicago.edu	Ok	It will start with the following print-out: <i>...Time in seconds: ...</i>
2	“user1” connects - server : champion.cs.uchicago.edu (10000) - port (TCP) : 50001 - port (UDP) : 50000 - nickname : user1	Ok	
3	“user2” connects - server : champion.cs.uchicago.edu (10000) - port (TCP) : 50001 - port (UDP) : 50000 - nickname : user2	Ok	At this point, the user list should show the other.
4	“user1” and “user2” start chatting	Ok	
5	“user1” sends a file to “user2”	Error /Ok	???
6	“user1” sends another file to “user2”		
7	“user2” sends a file to “user1”		
8	“user2” disconnects	Ok	Disappear in the list
9	“user2” connects again	Ok	Appear in the list, can chat again.
10	“user3” connects - server : champion.cs.uchicago.edu (10000) - port (TCP) : 50011 - port (UDP) : 50010 - nickname : user3	Ok	
11	“user1”, “Jake2”, and “user3” start chatting		
12	“user3” sends a file to “user1”	Ok	
13	“Jake2” exits by pressing “closing icon”	ok	To check closing correctly: disappear in the list
14	“user4” connects - server : champion.cs.uchicago.edu (10000) - port (TCP) : 50001 - port (UDP) : 50000		

	- nickname : user2		
15	“user4” sends a file to “user1”	Ok	
15	Kill “user2” process		To check if server is tracking correctly: disappear in the list soon (or RECV mesg 3 from server)

* Note : there might have been a little delay, but they worked well.

Comments

* We added some comments on how to handle the following requirements.

- **“Both the client and server program must be able to handle situations where clients terminate and restart.”**

“Client Status” checking message would cover this situation. The server periodically sends “ISALIVE” message to all clients and waits for the response for a specified period. If there is no respond from a client, the server would consider that client is terminated and send a “Remove_User” message to the rest of clients.

- **“You must take steps to make sure [UDP] communication is reliable.”**

To increase reliability, we choose the simplest way. Just send messages twice. We do not expect this redundant messaging mechanism work perfectly, but we expect it work reasonably. For example, we have never experienced the lost message after we introduced this mechanism.