CS162 Spring 2011 - Project 2

Instructor: Ion Stoica
Stephen Dawson-Haggerty, Jorge Ortiz, David Zhu

1 Project 2: Network Programming

In the first project, you were asked to design and implement the core server logic behind a chat server. In this project, you will design the network protocol that governs the interaction between the client and the server, as well as a text-based client that implements this protocol.

The goal of this project is for you to use the networking socket abstraction available to you to build a correct, parallel server program, as well as a client. This project is to be completed with your group. Similar to the last project, you will be submitting a design document prior to the final submission of the code.

All specifications and requirements described in project 1 carry over to project 2 unless otherwise stated in this or future version of this document. Announcements will be made when/if this document changes.

1.1 Server requirement

In Project 1, you implemented a server with basic chat functionalities. In this project, you are asked to extend the server with the capability to interact with a remote clients over a network. Using the socket interface, the server will accept connections from a remote client, interact with the clients, and terminate the connection. It is important to realize clients can disconnect at any time. Being able to gracefully handle disconnection and slow/unresponsive clients is essential to building reliable server system.

Client's request can come in asynchronously and unpredictably. For each user that attempts to log in, the server verifies the validity of the login request. If the server is not at maximum capacity, it allows the log in to proceed. If the server is at capacity, the user connection is placed on a waiting queue.

There are intentional and unintentional disconnects. Intentional disconnects (calls to disconnect from the client to the server) indicate the user's intention to withdraw from the system. Unintentional disconnect may result from the end user shutting down the client program, or disconnect the client from the network. Both types of disconnections should be handled. All state associated with the client should be cleaned up before actually removing the user from the system. Any error messages must be relayed back to the client, if the network connection between the client and the server is still functioning. Details on how these error messages should be displayed can be found in the section on *Clients*.

When the user is accepted into the system, the server shall respond to a client's request in a timely fashion. A delay in processing one client's request should not prevent other clients' request from being fulfilled. You should consider using multiple threads to improve your

server's performance. A mechanism that provide isolation between the client flows is key. This is especially important in a networked environment since network delay is often hard to predict, and requests can fail silently.

You are responsible for designing the protocol for client/server communication. It should be built on top of TCP sockets, and it should allow users to manage group membership, both unicast and group-broadcast messaging and receive messages from other users. The server must be able to forward errors to the client that it may encounter while processing the client's request. The protocol needs to be extensible since we will be adding more functionality in future projects.

The client can log out at any time. When a client logs out, the server should finish processing all of the outstanding requests for the client(both outgoing and incoming messages and errors).

Please continue to make calls to TestChatServer's logging functions as required by Project 1. We will be corresponding those with the output on the client side to verify the correctness of your implementation.

1.1.1 Timeout and Disconnection

The networking component brings several new challenges to the server. It is important that a client should not be able to hold on to the server's resource indefinitely without being admitted to the system. Thus **the server needs to disconnect clients that are connected but not logged in for more than 20 secs.**

In addition, the client can become disconnected at any time due to network error. **The server needs to detect this and automatically log off of the client – the server log should reflect this action.** In this particular case, all the messages sent to the disconnected client should be dropped (and logged). However, the server should finish sending any outstanding messages originated from the disconnected client.

At a high level we require the following:

1. Clients connect to and disconnect from the chat server at any time.
2. A user can choose to join multiple chat groups.
3. User and chat group names must be unique and concurrent reading of these names are allowed.
4. Message delivery must be in the order that is received. No two users in the same group shall see the same two messages delivered to that group in a different order.
5. Senders and receivers can log off or leave a group while a message is in transit and graceful error handling is expected.
6. A call to send/receive packet on behalf of one user should not prevent other users from sending/receiving.
7. Errors are relayed back to the remote clients through the network. Except in cases where the client has become disconnected from an unknown reason. In this case, errors should be logged by the server.
8. Interruption of network connectivity shall be handled gracefully. There should be no exception that does uncaught.

1.2 Client requirement

You will design and implement the client in accordance with your protocol design. For this part, you will implement a text based client that accepts commands from standard input, issue the command to the server, and output to standard output according to the following table.

Command	Type	Output	Description
`connect host:port`	sync	connect [OK, REJECTED]	Connect to a server with hostname and port
`disconnect`	sync	disconnect OK	Disconnect, invoke logoff if still logged on
`login username`	sync	login [OK, REJECTED, QUEUED]	Login to the system with username, can be rejected because of duplicated user names.
`logout`	sync	logout [OK]	Log out of the system.
`join gname`	sync	join gname [OK_CREATE, OK_JOIN, BAD_GROUP, FAIL_FULL]	Create a group with the specified name if it does not exist, otherwise join an existing group.
`leave gname`	sync	leave gname [OK, BAD_GROUP, NOT_MEMBER]	Leaves a group or return error if the group does not exist or the user is not a member.
`send dest sqn "msg"`	async	send sqn [OK, BAD_DEST, FAIL]	The sender gets a confirmation or error because of an invalid destination or full buffer.
—		sendack sqn FAILED	The sender gets an error message, if there is a failure while delivering the original message to all the destinations.
—		receive sender dest "msg"	When the client receives a message, it prints the message to the console.
—		timeout	The server has disconnected the user because of inactivity.
`sleep time`	sync	sleep time [OK]	The client sleeps for a specified number of milliseconds. During this time, no command shall be sent, but incoming messages are processed.

There are two types of commands, synchronous and asynchronous. When a client issues a synchronous command, it will wait for the final result before proceeding to process the next command. This is not true for asynchronous command such as `send`. `send` would receive an initial acknowledgement if the destination is bad or the queue is full. However, some time later, it is possible for the client to receive a message indicating a message did not reach all of its intended destinations. When that happens, the client should output a `sendack` message.

Another unusual command is `sleep`. Normally, the client would process all the commands that are in the standard input stream. The `sleep` command forces the client to take a specific amount of time to process this particular command. We will be using this to control timing of events in our tests.

When the client is printing the response, it must follow the format shown in the above table. The strings enclosed in brackets are possible options. For example, connect [OK, REJECTED]

means that the output can be either connect OK or connect REJECTED.

1.3 Client Server communication

The client and the server communicates over a network connection, represented through a socket interface. Since you cannot send Java objects directly through the socket interface, you have to develop a scheme to encode all the necessary information in the network packet. Upon reception, the server/client need to reconstruct the necessary objects using the encoded packet that is received. It is important to clearly document this encoding scheme in your design document.

1.4 Design Document

Your design document should include both the modifications to your server to adapt to a networked environment and the design of your client. In the server portion, the document should state clearly which threads are responsible for sending and receiving of messages, how each message travels through the entire server and to be delivered out later. It should also include the failure cases you anticipated and how your system will handle them. You are required to draw a state diagram showing the transition between different server states and any output as it receives commands from the client.

On the client side, you are also required to submit a state diagram depicting client's transition between different states. Finally, your design document should also include a detailed description of the protocol the client and the server uses to communicate with each other. This description should include the message format and how requests and responses are encoded. It should also include the steps involved for a typical action of sending message.

In addition, the document should include a test plan for the server, the client and the interaction between them.

1.5 Submission and Testing

Our tests for Project 2 will be based on your client and your server. We will create instances of your client and provide a list of commands to be processed by your client. We will verify that your client produces expected output. The testing process will look similar to the following.

```
$user> java ChatServer 8080
$user> java ChatClient < inputforChatClient1 >output1
$user> java ChatClient < inputforChatClient2 >output2
```

We will be creating many concurrent `ChatClients` and one instance of the ChatServer. Make sure your main method of the server and client takes parameters as specified above. Our commands for each client will consists of the following steps/aspects:

1. Attempt to connect to the server.
2. Either timeout and get disconnected or log in with a valid or invalid username.
3. If the user successfully logs in, the user query for a list of users and groups.

4. The client will verify the list of users and groups are valid.
5. The client will try to join a valid or invalid group.
6. If successful, the client will send a message to the group.
7. The client will send a message to a valid or invalid user and check for possible failure.
8. The client will log out normally or simply disconnect at any point.

Updates

As this is a work-in-progress, we will be updating this document as we answer questions on piazza. This collects important updates.

Version 1.1

1. Added a BAD_GROUP to joingroup command.
2. Made the command line parameter optional for ChatClient

Version 1.2

1. Fixed return code for login and connect. Users are now queued after they attempt to login rather than connect.
2. Fixed the time out to be 20 secs.
3. Added an error message to be returned if there is a timeout instead of a disconnect.