KTN ELM CHAT

Documentation

By Group 61

March 2, 2014

Kristoffer Larsen, Mari Bergendahlv, Heidi Halvorsen,
Olav Markussen, Julie Vanessa Skåtun and Raymi Eldby

Contents

Introduction

KTN ELM CHAT is a task assigned to a group of students at NTNU by the staff of the course TTM4110 – Communication Services and Networks. The task is described in two documents, provided here. The plan for the project was to be delivered due the March 3rd of 2014, while the actual implementation was due March 24th of 2014.

In essence, we were instructed to create any chat application consisting of a pair of classes (Server and Client) that implemented a given protocol. As a bare minimum, the chat needed to implement the following functionality:

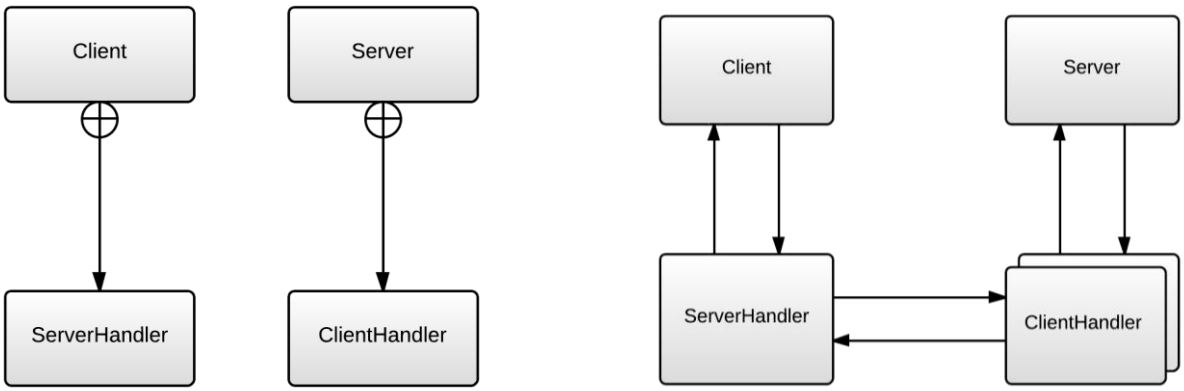
- Socket connection
- Login with username
- Messaging
- Logout
- Multiple clients connected simultaneously

Our implementation was done in Java, providing two jar files: Client.jar and Server.jar. The protocol was implemented using Java's built-in classes for socket connections as well as the JSON libraries provided by json.org.

None of the group participants are experts at Java, we will not be held liable for any damages inflicted upon executing the code we have developed. Use this code at your own risk.

Basic Structure

The chat consists of four classes, the Client with the inner class ServerHandler, and the Server with the inner class ClientHandler. The Client and Server classes are ideally independent of the protocol, in the sense that no protocol operations are carried out by themselves. These operations are to be handled in the ClientHandler and the ServerHandler.



In terms of data-flow, data can only originate in the Client and Server classes. All requests the ClientHandler send originate in user input (handled in Client), and all responses are based upon data from the Server. However all data transaction need to be done through the handlers, the Client and Server will never directly communicate.

The classes are explained more in depth in the next section – classes, and the data flow are explained further in the protocol sequence diagrams section.

Classes

Client

In essence, the client needs to be able to take input from the user and convert it into appropriate commands. These commands are then passed to the Server through the ServerHandler.

Below follows two tables that list core variables and methods for implementing the required functionality. Other variables and methods may be added in order for the core methods to be carried out better or more orderly.

Field Name	Type	Description
HOST	String	Holds the host IP address.
PORT	int	Holds the port to use for TCP connection.
serverHandler	ServerHandler	Carries out protocol, when the user types into the console, the interpreted command should be called on this object.
in	BufferedReader	Reads user input through the console.

Method Name	Parameters	Output	Description
getKeyboardInput	None	String	Waits for the user to submit a string to the console and returns it.
interpretInput	String	void	Takes a string and attempts to interpret is as a command. If it recognizes the string as a command, it calls upon the corresponding command in the ServerHandler.
pushMessage	String	void	Pushes a string to the user through the console.

ServerHandler

The ServerHandler carries out all requests given by the Client by using the provided protocol. It wraps the requests into JSON objects and send them to the server through a socket. Also, it needs to receive responses (formatted as JSON objects) from the server and if necessary push them to the Client.

The ServerHandler runs as its own Thread, the reason for this is that it needs to listen to its socket at all times. Because the Client is already using the daemon Thread for listening to the console, a new Thread must be created in order to also listen for responses from the server.

Below follows two tables that list core variables and methods for implementing the required functionality. Other variables and methods may be added in order for the core methods to be carried out better or more orderly.

Field Name	Type	Description
socket	Socket	The socket used to communicate with the server.
in	BufferedReader	Reads data from the socket.
out	PrintWriter	Write data to the socket.

Method Name	Parameters	Output	Description
getResponse	None	JSONObject	Waits for the server to push a response and returns the response as a JSONObject.
handleResponse	JSONObject	void	Takes a JSONObject and unpacks it in order to decide if- and how the response should be pushed to the Client.
request[Command] (e.g. requestLogin)	Various types of parameters	void	Carries out requests made by the Client by creating JSONObjects and sending them to the server. The input varies, as different commands require different input on server side.
sendRequest	JSONObject	void	Sends a JSONObject to the server through the socket.

Server

In essence, the server needs to be able to do three things. Firstly, it needs to be able to accept connections and assign the connection to a ClientHandler. Secondly, it needs to push messages received to all logged in clients. Thirdly, it needs to hold a log of all messages sent to the server, in order for it to be provided to any clients logging in.

Because the protocol states that a request to log in with a username already taken should be responded to with an error, the Server must additionally provide a method for checking if a username is taken.

Below follows two tables that list core variables and methods for implementing the required functionality. Other variables and methods may be added in order for the core methods to be carried out better or more orderly.

Field Name	Type	Description
PORT	int	Holds the port for listening to incoming connections.
serverSocket	ServerSocket	Provides functionality for accepting new TCP connections.
clients	List<ClientHandler>	Holds all the ClientHandler objects for reference. This is needed in order to broadcast messages and to check for taken usernames.
messages	List<String>	Holds all the messages sent to the chat for backlogging.

Method Name	Parameters	Output	Description
getClient	None	Socket	Listens for incoming connections and returns the socket for the connection.
pushMessage	String	void	Adds a message to the list of messages and then pushes it to all logged in clients.
getMessages	None	List<String>	Returns a list containing all messages in the chat.
isAvailableUsername	String	boolean	Checks if the given username is taken. Returns false if the username is taken, and true if not.

ClientHandler

The ClientHandler handles the connection to a client, carrying out the protocol. It listens to the socket for requests from the client and uses the data provided by the Server to make create an appropriate response.

Additionally, it provides functionality for the Server class to get the username of the client connected through the ClientHandler, as well as functionality for sending a message to the client (for broadcasts).

The ClientHandler runs in its own Thread. This is because each ClientHandler object needs to listen to its socket at all times for requests from the client. Also, the Server is already using the daemon Thread for listening to new connections.

Below follows two tables that list core variables and methods for implementing the required functionality. Other variables and methods may be added in order for the core methods to be carried out better or more orderly.

Field Name	Type	Description
socket	Socket	The socket used to communicate with the client.
serverSocket	ServerSocket	Provides functionality for accepting new TCP connections.
clients	List<ClientHandler>	Holds all the ClientHandler objects for reference. This is needed in order to broadcast messages and to check for taken usernames.
messages	List<String>	Holds all the messages sent to the chat for backlogging.

Method Name	Parameters	Output	Description
getRequest	None	JSONObject	Listens for incoming requests from the client and returns it as a JSONObject.
handleRequest	JSONObject	void	Takes a request and creates a response for the client in accordance with the protocol. The response is then sent to the client.
getUsername	None	String	Returns the username assigned to the client.
sendMessage	String	void	Packs the message in a JSON object (using the protocol) and sends it to the client.

The Protocol

The protocol is based on interchanging messages called requests and responses. An array of different requests are available to the client, currently restricted "login", "logout" and "message". Whenever the client sends a request, the server needs to send a response, the request name should then be used as the response name.

Responses usually hold a status, for example a requested login may return either with status “OK” if the username provided was granted, or with status “error” if it was not. In addition, contextual data may be provided, errors usually need to provide context in order for the user to understand exactly why their username was rejected.

In most cases, the server only sends data when a request is received. However, when a new message is received by the server, the server needs to broadcast this to all users. In order to do this, a response is generated without any request, this type of response is called a "new message".

Sequence diagrams are provided to give an impression of how the protocol works:

Sequence Diagrams

The sequence diagrams provided show how the protocol *may* be implemented, some parts are still up for changes. In the diagrams a certain formatting is used:

- Names of methods used are shown above the flow arrows, while arguments are listed below.
- If the method returns a value, it is passed inside a new flow arrow back to the place where the method was called.
- Note that requests and responses are in reality passed as strings, the values are formatted into the string using JSON.

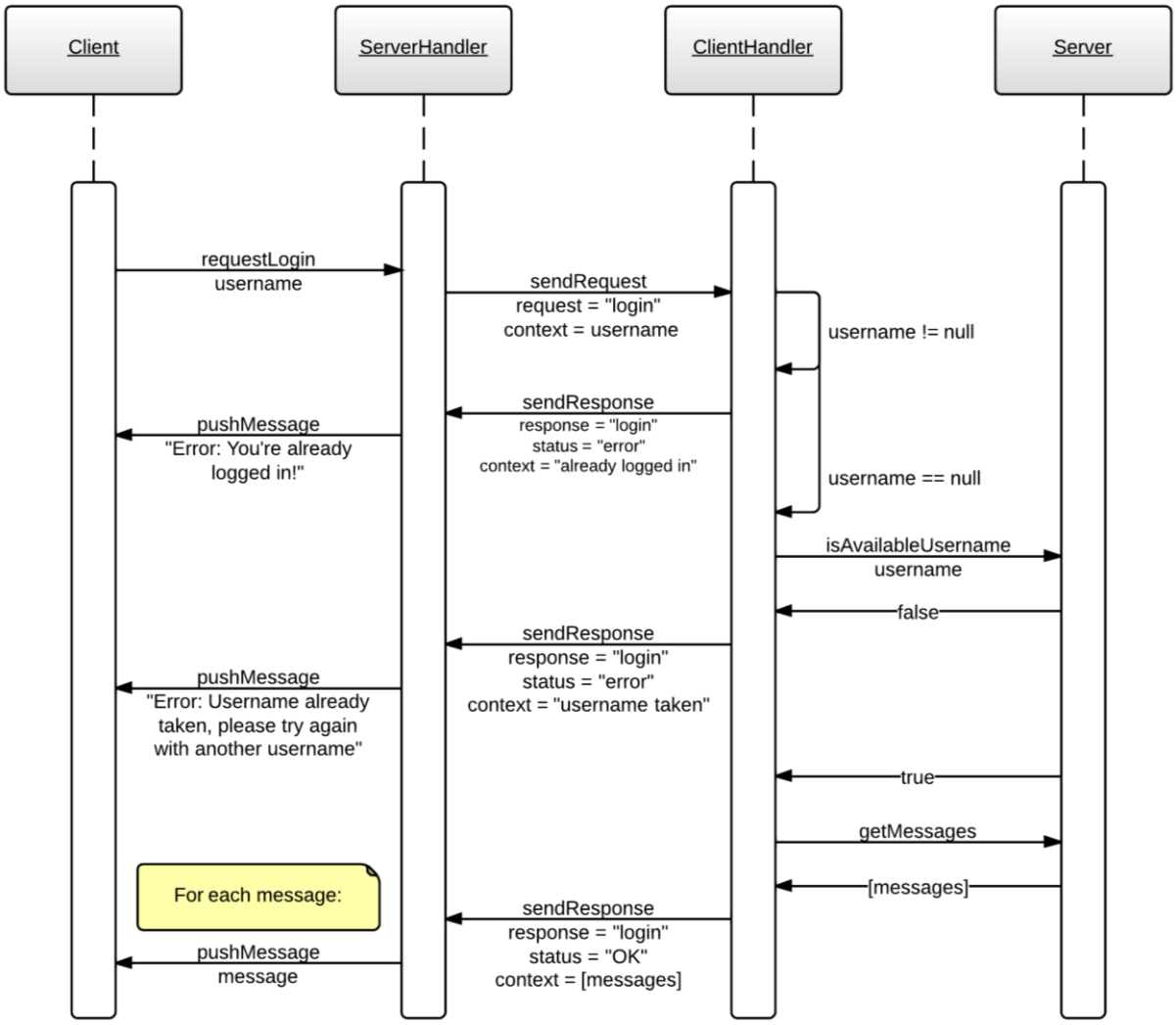
Login sequence

The login sequence is initiated by the Client by calling the requestLogin method on the ServerHandler. A request is sent to the ClientHandler using the request “login” with the desired username as context.

The ClientHandler then checks if any username has been assigned to the client. If it has, it means the client is already logged in, and we need to create an error response notifying about how the login was rejected. The ClientHandler pushes the error.

If the user was not logged in, the ClientHandler checks with the Server to see if the username is available. If it’s not, an error response is created and sent to the Client, and the ClientHandler pushes the error.

However, if the username is available, a response with status “OK” is created and sent along with a list holding all the messages (a backlog) as context. Each of the messages are then pushed to the Client by the ClientHandler.

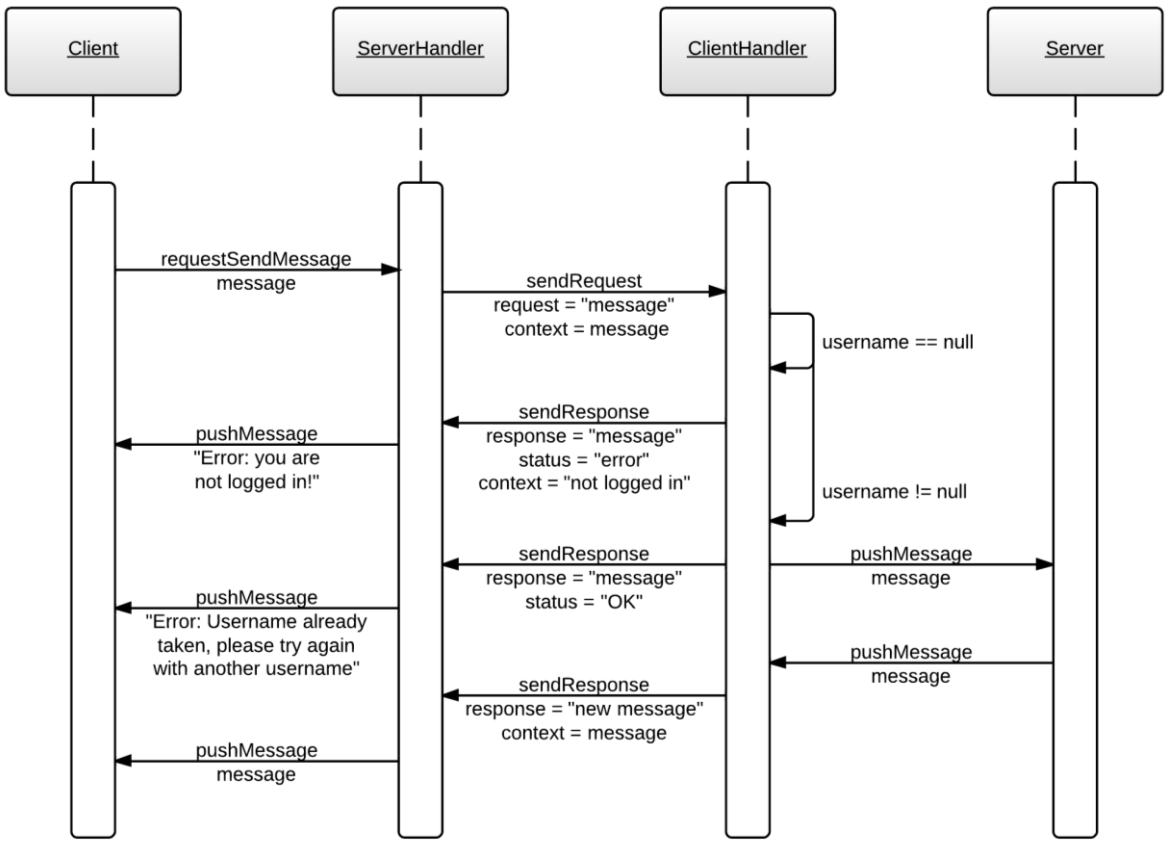


Message sequence

The message sequence is initiated by the Client by calling the requestSendMessage method on the ServerHandler. The request is then sent to the ClientHandler as a “message” request, using the message as context.

If the user is not logged in, no username has been assigned to the client, in which case the ClientHandler immediately returns an error response.

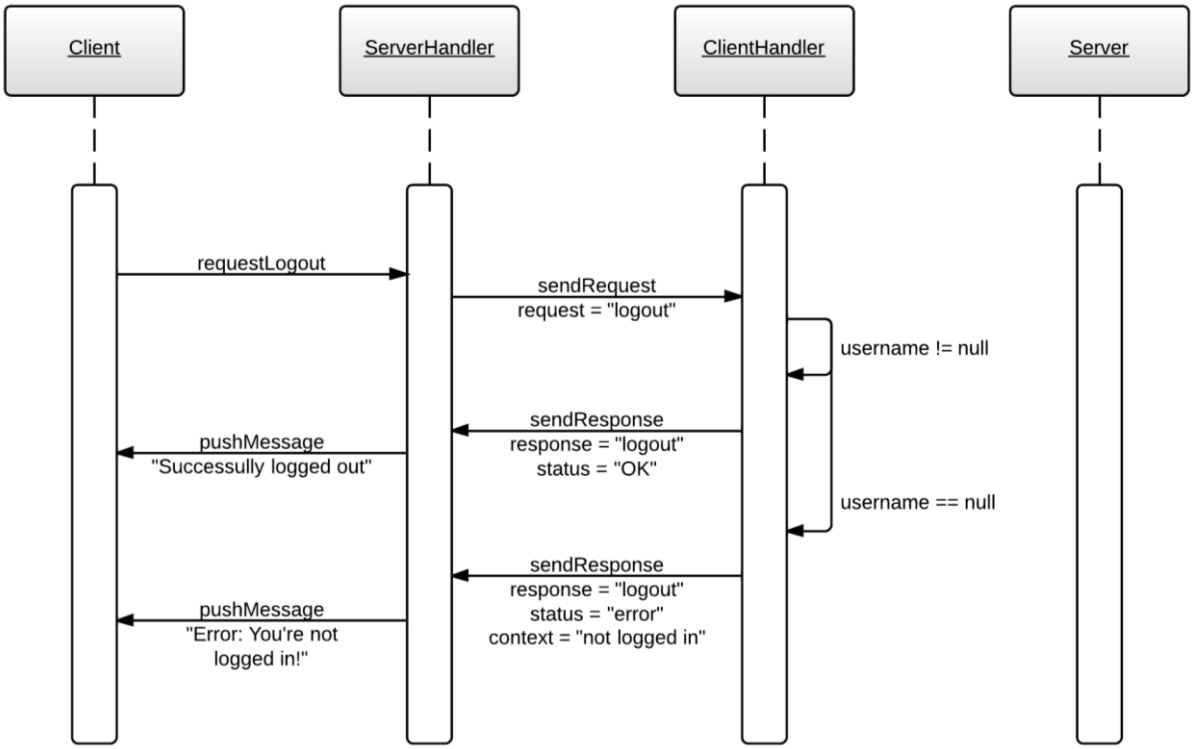
However, if the user is logged in, the ClientHandler creates a response with the status “OK”, before it pushes the message to the Server. The Server then pushes the message to all ClientHandler objects given that they are logged in, this of course includes the initiating ClientHandler. This triggers the ClientHandler to send a new response, called “new message”, which carries the message as context.



Logout sequence

The logout sequence is initiated by the Client by calling the requestLogout method on the ServerHandler. The request is then sent to the ClientHandler as a “logout” request.

If the user is logged in, a username has been assigned to the client, and the ClientHandler creates a response with the status “OK”. However, if the user is not logged in, the ClientHandler creates an error response with the context “not logged in”.



JSON formatted requests and responses

Here are the fields and values of all JSON formatted requests and responses.

Request \ Field	request	context
login	“login”	username
send message	“message”	message
logout	“logout”	None

Response \ Field	response	status	context
login granted	“login”	“OK”	[messages]
login rejected (username taken)	“login”	“error”	“username taken”
login rejected (already logged in)	“login”	“error”	“already logged in”
message sent	“message”	“OK”	None
message rejected	“message”	“error”	“not logged in”
logout granted	“logout”	“OK”	None
logout rejected	“logout”	“error”	“not logged in”
new message	“new message”	None	message