CSE 4481

TERM PROJECT

PHASE 3

PROFESSOR: MARK SHTERN

DUE DATE: 04/01/2013

GROUP MEMBERS:

ARFEEN, OSMAN
MAHESH, NIVEDITA
TRAN, ALEXANDER

CONTENTS

1. ABSTRACT……3
2. DEFINITIONS……3
3. SERVER DESIGN……4
4. CLIENT DESIGN……8
5. GUI……10
6. TESTING……27
7. SECURITY ISSUES……30

ABSTRACT

The ultimate goal of this project is to build a chat application that enables communication over a public network. The users of this application are users from a non-technical background for online support. The application supports two kinds of users: anonymous users (Customers) and help-desk users (HD Agents). The high-level design of this project comprises of three main components: Server, Client library and a Client Graphical User Interface (GUI). The client GUI provides for a simple user interface that is user-friendly and can be used by people without requiring extensive knowledge of the underlying logic. The client library, as the name suggests, serves as a library or an API for the client GUI to use. In order to serve the requests made by the GUI, the client library interacts with the server, which acts as a central point of authority that works to enable communications between clients. The decision to separate the project into three separate parts was made so that the different components could be built separately independent of the other parts. This makes the code every efficient and versatile, since it can re-used to build another application with just a change in the GUI component.

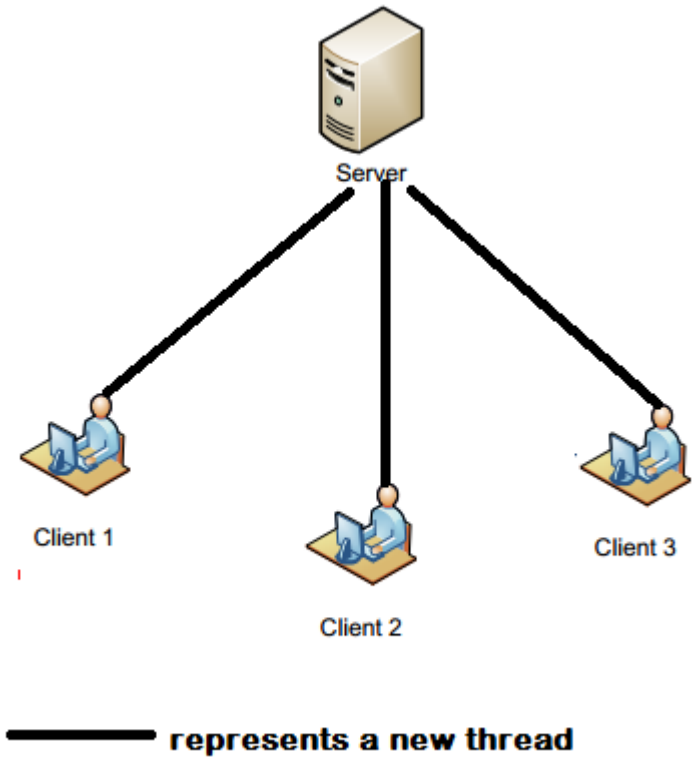
DEFINITIONS

1. **Customer:** An anonymous user is referred to as a customer.
2. **HD Agent:** A helpdesk user is referred to an HD Agent.
3. **Client:** In this project, a client refers to a client machine and not the user.
4. **Client Context:** Client Context is an object that is capable of holding socket information and insensitive user information such as username, ID, display name etc.
5. **Message Handler:** A class that is responsible for the deserialization of messages.

THE SERVER DESIGN

THREADING

- **Threading in the server is handled in such a way that it creates a separate thread for every client when a new socket connection is opened**. This way, each client has its own thread and thereby, avoids any kind of ambiguity. The following figure demonstrates the point.



- Every socket is associated with an object called the Client Context. It holds three kinds of information: Details about the socket, insensitive information about the user (in the case that they are logged in) such as username, ID, display name and information about the Client Context of the user at the other end.

- Before a user can log onto a client machine, a ping request is sent to the server to check if it is alive. If it is, a socket connection is established and the Client Context object is created. However, it does not hold user information yet. At this point, the user is allowed to login and if authenticated, the user information is embedded onto the Client Context. **We have chosen to associate the socket with the Client Context instead of directly creating an association with the user (For example: User Context Object) because it allows the user to logout and yet keep the socket alive.** This way, the socket can be used by another user. If we had done otherwise, the user logging out would have meant disruption of the socket connection.

HOW INCOMING MESSAGES FROM THE CLIENT ARE HANDLED

- As we know, each client is handled in its own thread. **Within each client, for each incoming message, a new thread is created and it is handled in there.** If we had designed the server in such a way that it would handle all the messages in a single thread, then it would have resulted in negative consequences. For example, it could have brought the server down and even have the user logged out from the client machine. In order to avoid such situations, the above decision was made. This way, even if a thread dies, all the other threads are still active and not affected by it.

- **JSON:** All incoming messages received by the server are in JSON format. Not just incoming messages, even the outgoing string messages are wrapped in a JSON object. We made the decision to not just plainly send and receive string messages; instead use JSON to format because parsing string messages

can prove to be quite messy and inefficient. JSON on the other hand provides for easy deserialization of messages. To further simplify the parsing process, we used Google GSON; a library that converts that JSON to java objects and vice-versa.

- Deserialization of incoming messages is done by a Message Handler (See Definition). Below is an example of some of the patterns that were used to decipher and parse messages in JSON format.

PING

Client -> Server
Server-> Client
{

"ping":true

}

AUTHENTICATION REQUEST

Client -> Server
{
"authenticationRequest":
{
type: int
username: string
password: string
user: string
}
}

HD Agent List Request

Client -> Server
{
"HDAgentListRequest":true
}

END CONVERSATION

Client -> Server
Server-> Client
{
“endConversation”:true
}

Once the message is deserialized and the type of message is deciphered, the message is sent to the appropriate class that is responsible for handling that type of message. Based on the type of message, the server creates an appropriate response and sends it to the list of outgoing packets. The outgoing packet holds the socket it needs to send the message on and the message itself. **We use a common list to send all the outgoing messages because this way it allows the server to send many messages at a time.**

The server makes use of two main data structures. Firstly, a queue that holds a list of Client contexts of users. To be specific, they are anonymous users have do not have an HD Agent to serve them yet. Such users are placed into this queue. Secondly, the server holds a collection of active HD Agents. The collection in this case is a hashmap that maps the ID of the HD Agent to the Client Context.

THE CLIENT DESIGN

THREADING

The client handles threading in a way very different from the server. The server as we know primarily acts as an object that responds to messages. It never initiates a new message or conversation. And, therefore, the server is capable of handling numerous clients and messages simultaneously. In other words, it can handle multiple threads at the same time

However, the client is designed to talk to only one server. The client, unlike the server is open to 2-way communication. The client, both receives incoming messages from the server and sends new messages. The client creates a new thread for every message it sends and receives. At any point in time, the client only deals with one thread. This implies that regardless of the number of messages that are waiting to be received or sent, the client shall only process one at a time. The rest will have to wait to be processed.

SERVER-CLIENT LIBRARY-GUI

When the client is looked at in detail, it comprises of the Client library and Graphical User Interface (GUI). When a user first interacts with the application, what he sees first is the responsibility of the GUI. Depending on the actions of the user, the GUI uses the Client library to generate an appropriate response. The Client library is where all the underlying work is

done. Starting from authenticating a user to sending to message to another to logging out, the client library handles all of it.
We have designed the Client library to mimic an Application Programming Interface the GUI can use. Depending on the requests of the GUI, the Client library interacts with the server. The Client library can be imagined to be a sort of middle man between the server and the GUI.

HOW MESSAGES ARE HANDLED

For incoming messages, we have developed an Interface called GuiGlue. GuiGlue acts as a sort of middle class between the Client library and the GUI. When the client library receives a message from the server, it processes it based on the type of message. The way it processes it is very similar to how the server processes its incoming messages. As previously discussed in the server design, all messages are JSON formatted. Like in the server, the client has its own Message handler class to deserialize the JSON string.

Once deserialization is complete and the type of message deciphered, it is sent to the appropriate processing class. To ensure clean code, we created a separate class to handle each type of incoming message.
After the message is processed, the client now has to communicate the results to the GUI using a class called Controller that implements The GuiGlue interface. The interface is built on top of the library.
The GUI details are discussed in the following section.

GUI

I. Outline of Design Requirements

GUI should be simple and user friendly. The GUI should not be cluttered with buttons and be straight forward to use for both the customers and the help desk users.

In the customers' case, they should be able to successfully navigate the GUI to receive help Also, they should not have extra, unncessary features. Certain messages should be displayed to help guide the customer to receive help from a help-desk user.

In the help-desk users' case, they should not be showered with hundreds of buttons on the GUI. Having too many buttons will impede a help-desk user's ability to use the GUI and help a customer efficiently. Instead, the help-desk user should be given a few buttons that are crucial for their operations, such as an ability to transfer over a customer to another help-desk user.

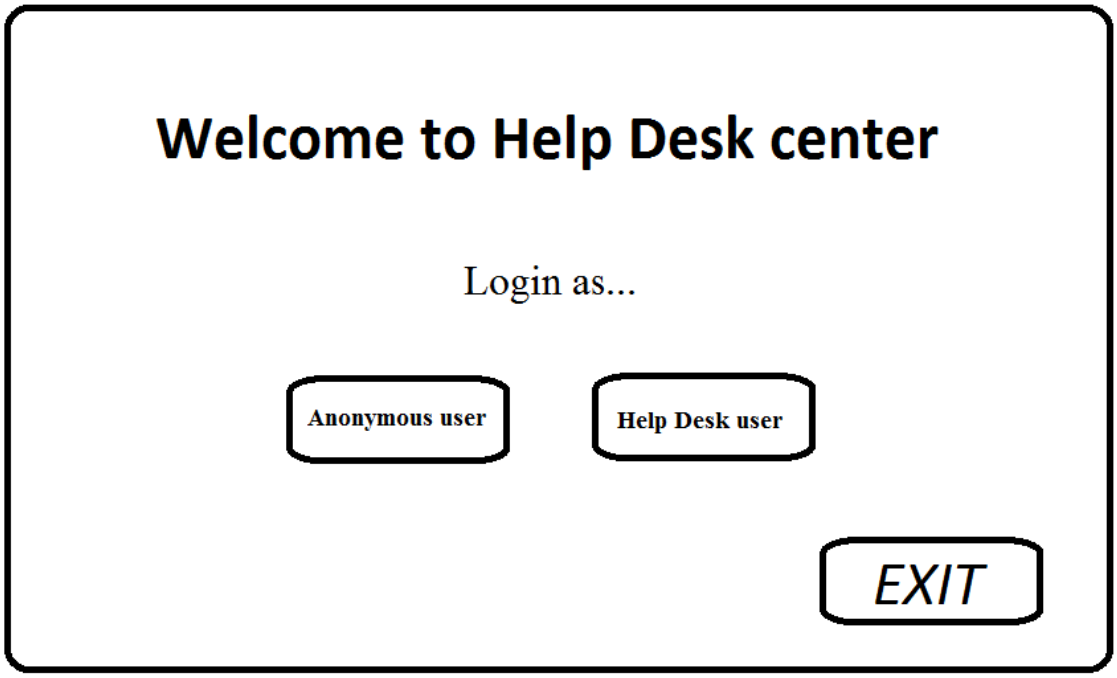
GUI should be able to display of all logged-on help-desk users after help-desk user authenticated. After a help-desk user successfully logs in, they should be able to see a list of all online help-desk users. This list should be updated in real time whenever a help-desk user logs in and out.

II. Contraints and Considerations

Time constraints and programming difficulty need to be considered. Java has a robust library for implementing a GUI. However, programmers have to consider how much time they have to learn and implement a GUI for their program. A nice looking GUI program with animated pop-ups might take considerable amount of time to program but still provides the same amount of help service to customer.

III. Proposed Designs

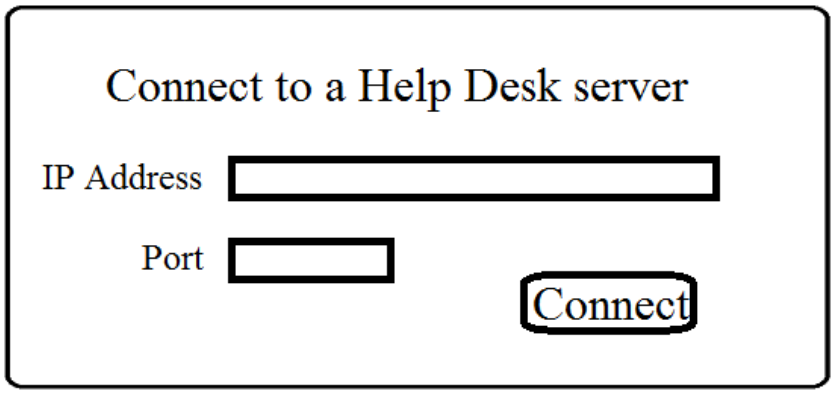
Using basic java swing components to build the GUI. The GUI will be built on basic swing components such as panels which consists of a list, text fields, text areas, labels, buttons. These some of these components will have a listener to implement interactivity with the user of the program. For example, adding a listener to a Login button will log the user in when the button is pressed.



Welcome Screen

Requirements: Since there are two types of users, the GUI should be able to forward the users to the appropriate screen. A message should be shown to welcome the user to the Help Desk center and another message that tells the user to log in as either Anonymous or Help Desk user.

Program design: Labels are used to display the welcome message and instructions. Listeners should be added to the appropriate buttons according to their specific functions. Logging in as a Anonymous user should forward the user to an available Help Desk user. Logging in as a Help Desk user should forward users to a login screen. The exit button exits the program and closes the display.



Previous Welcome Screen

Requirements: Users must be able to change the server settings, so a field for server address and port is needed. Also, users need to be able to connect to the server by the use of a button.

Program design: A listener is added to the connect button to take in the IP Address and Port text fields and use those values to connect to the server. Labels are used to denote what text field are what.

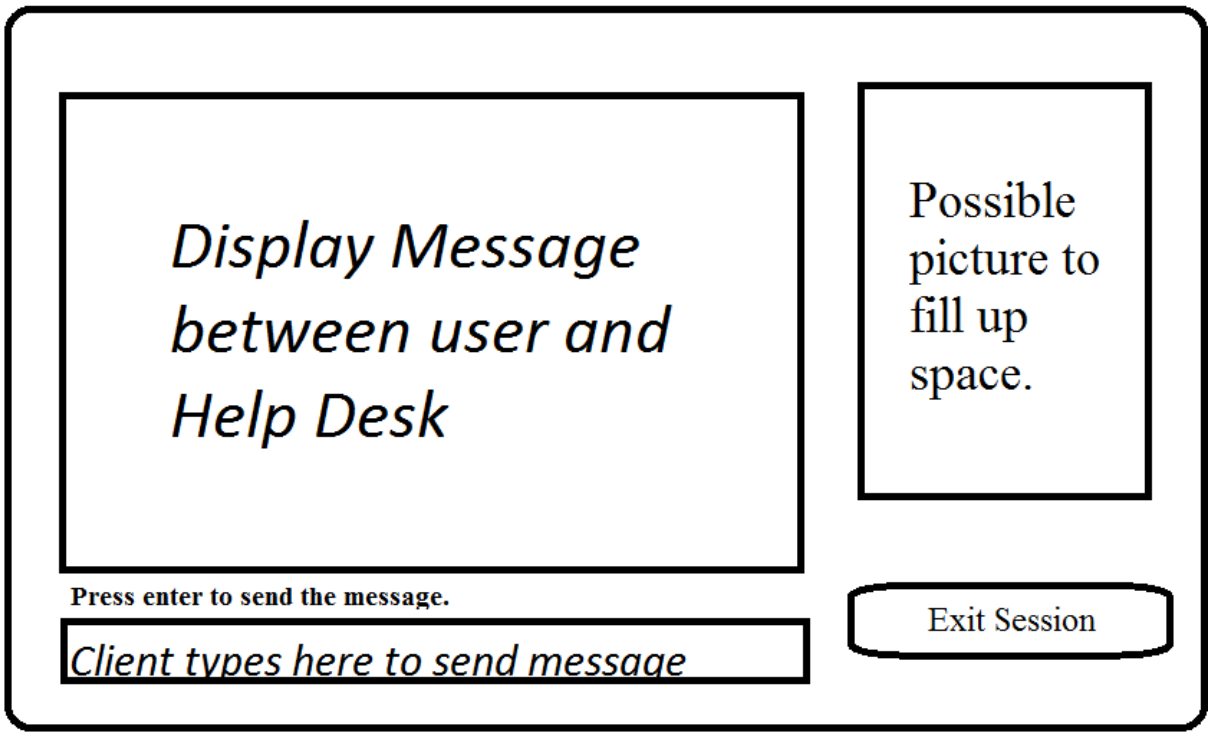
Why change the Welcome screen? Initially, this screen was the first screen that shows up when the program is launched. Later, this is moved to a setting screen that is invoked by a button on the final version of the Welcome screen. The reason for this action is that the customers might not know what they are doing, and might put in the wrong IP Address and Port to connect to the server. Thus, it will hinder the customer's ability to receive help.

Waiting screen

Requirements: The waiting screen should tell customers that they are being transferred to a help-desk user. Customers who are tired of waiting should be able to quit the application by the use of a button.

Program design: A label should be used to tell customers that they are being transferred. An exit button with a listener, when clicked on, should exit the waiting screen.

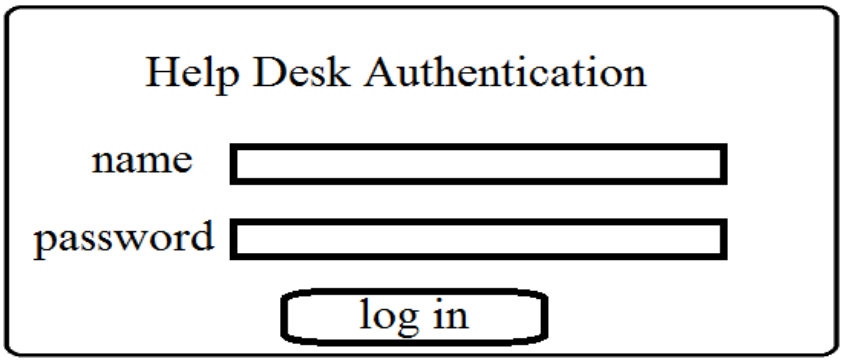
Customer Chat Screen



Requirements: On the customer chat screen, customers need to be able to send messages to the help-desk user and also receive messages. On the GUI, there should be a message that tell the user how to send message to the help desk user. Also, there should be a way for the user to quit chatting with the help-desk user.

Program design: A text area will display the messages the customer sends and also the messages coming from the help-desk user. A label will show how the customer will send their messages to the help-desk user. The text field(has a listener) will take in the customer's message and will send the message each time the customer presses enter.

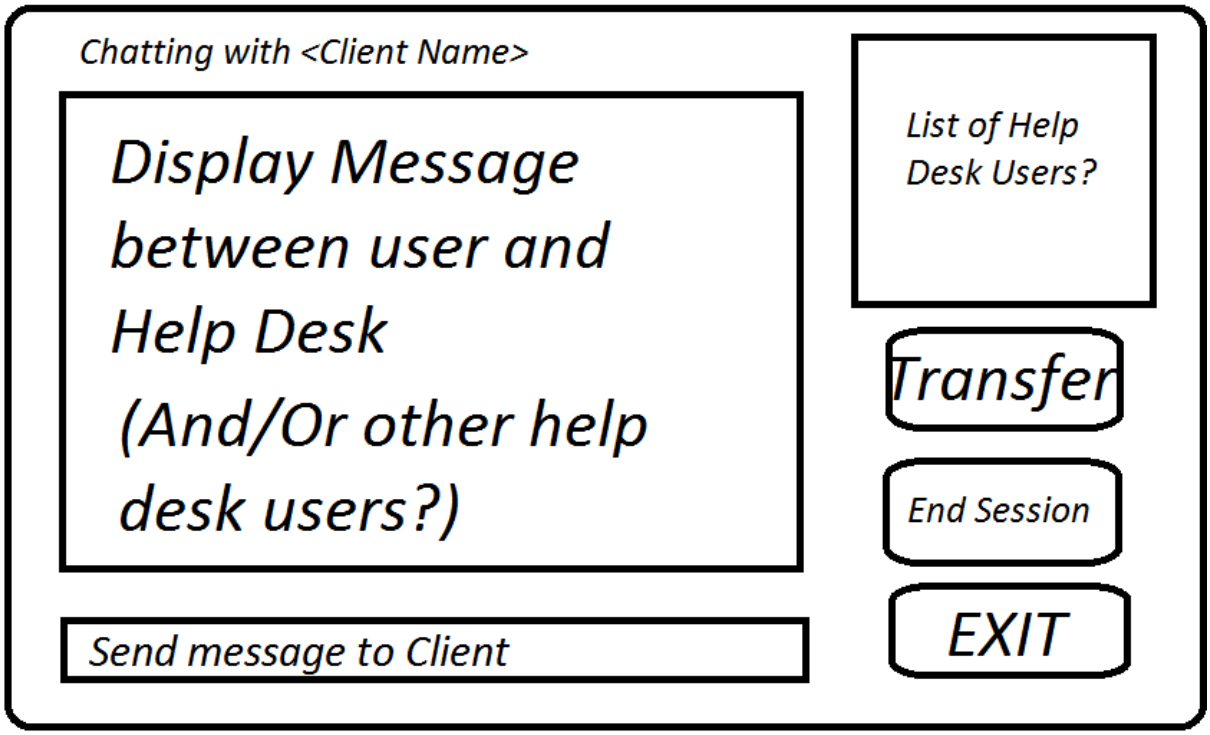
Help Desk Login Screen



Requirements: In order for the help-desk user to authenticate themselves, they must enter their names and passwords into the two fields. They then either press enter or press a button to log in with their user information. Also, to let the help-desk users know that they are on right screen, display a message that this is a help-desk login screen.

Program design: A label for showing that the user is at the help-desk screen, and a label for the user name and password text fields. Listeners are attached to the text field and button so that whenever users press enter or click the button, they are logged in.

Help Desk Chat Screen



Requirements: Help-desk users should be able to transfer customer to another help-desk, end customer session, exit, and chat with both a customer and help-desk users. Also, help-desk users should be able to see other online users when logged on.

Program Design: Very similar to the customer chat screen but with more function. A list will be used to show online help-desk users. This list will be updated whenever a help-desk user logs in or out. A button for transfer, end session and exit will be implemented with listeners according to their function. To use transfer, a help-desk user must be highlighted in the list. Also, double clicking the user in the list will bring up another window similiar to the customer chat screen which is the HD to HD chat screen.

IV. Finalized Designs

Welcome Screen

New changes from proposed design:

Added an option for user to choose a display name. This gives customers an option to create an identity for the help-desk users to call by their names. It would be odd for a help-desk user to say something like "Hello, anon342124341, how are you?" Also, the text field focuses every time the program is executed. This helps users so that they do not have to click in the text field to type in their display names.

Simplified Get Help! button. Instead of the previous "Anonymous user" button, a "Get Help" button is better suited for the task at hand.

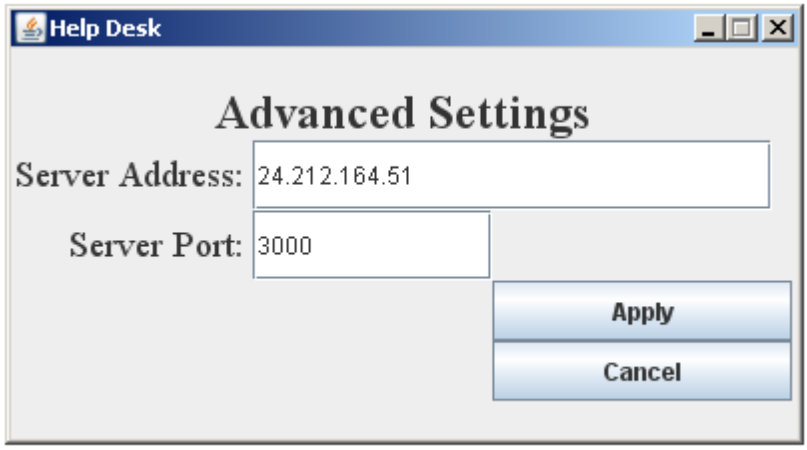
A user, who wants to get help, sees the button and without thinking, they will press it.

Removed the help-desk button and instead, use a listener to listen on key presses to bring up the help-desk login screen. The keys to press (at the same time) is Ctrl + Shift + H. The reason for this change is to hide information that customers do not need. Hiding the help-desk login button also makes it harder for malicious users to exploit the program, such as trying to find out a help-desk user's password.

Added Advanced Settings button so that any user that wants to change the connection settings can do so by pressing this button.

Removed the Exit button. The user can simply press the x button on the top right to close the program. This change simplifies the welcome screen even more.

Connection Settings Screen



New changes from proposed design:

Automatically loads the current settings into the text fields. If the settings were left blank, a user might accidently apply the changes while the Server Address and Port fields are blank. They are unlikely to remember the previous settings and will not be unable to connect afterwards. Of course, the users can just reopen the application to get the default settings back.

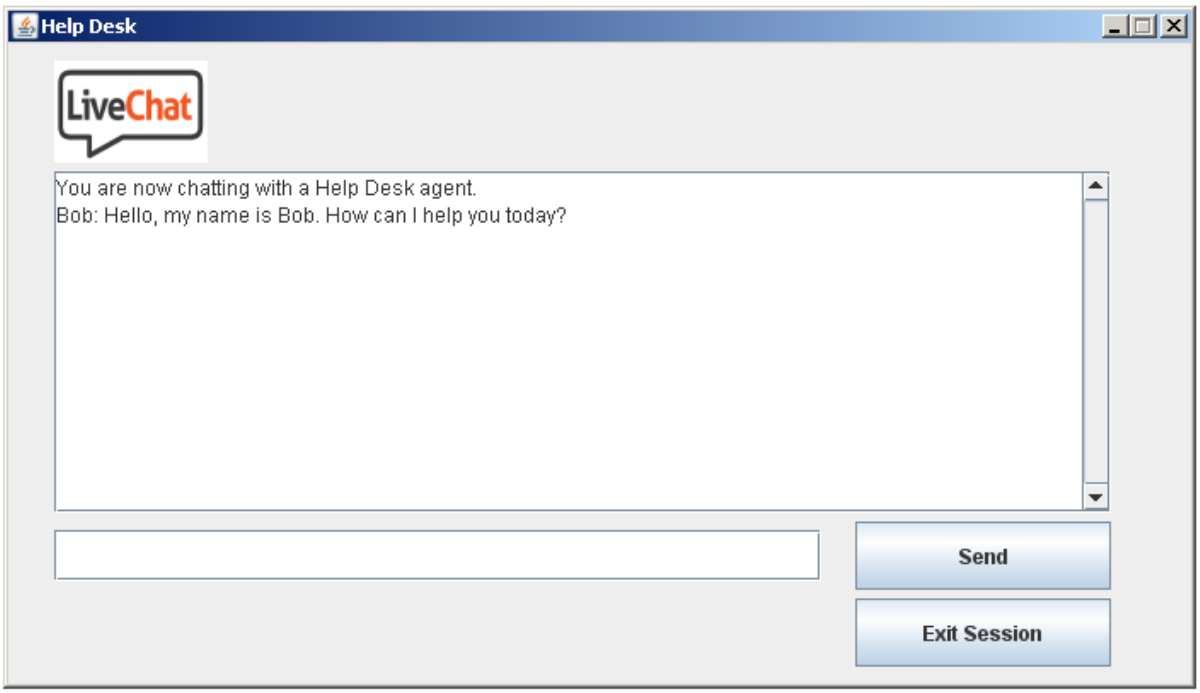
Waiting Screen

New changes from proposed design:

Better description in displayed message. The previous description "Transferring to <Help Desk user name>" was incorrect since the customer is not waiting on being transferred, but rather waiting on help-desk users to be available.

Exit button changed to "cancel" and moved to center. Instead of exiting the program, the user now has the ability to go back to the welcome screen when pressing the cancel button. The button is moved to the center to make it more compact and simple.

Customer Chat Screen

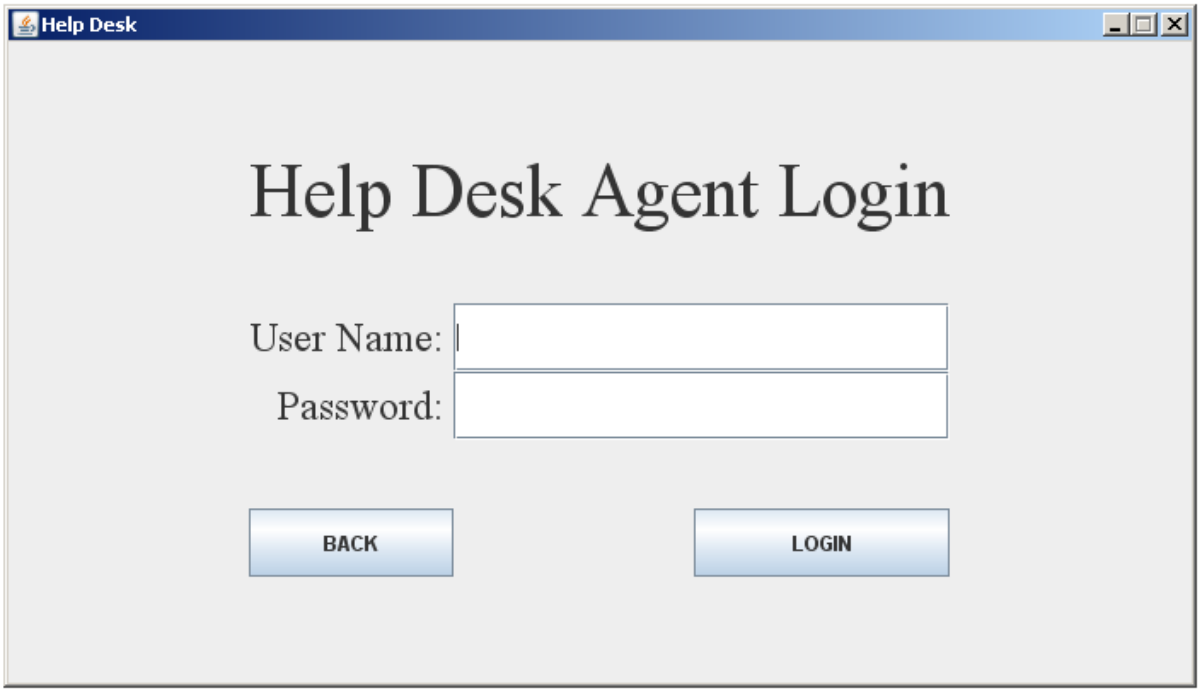


New changes from proposed design:

Removed the "press enter to send the message" message from the screen. Most chat programs have a send button to send their typed messages to the other user. Adding the Send button makes the "press enter" message redundant, and removing it helps compact the text field and text area

Changed picture location and expanded text area display. Changing the picture location makes it possible to expand the text area so that the user can see more of the messages.

Help-Desk Login Screen



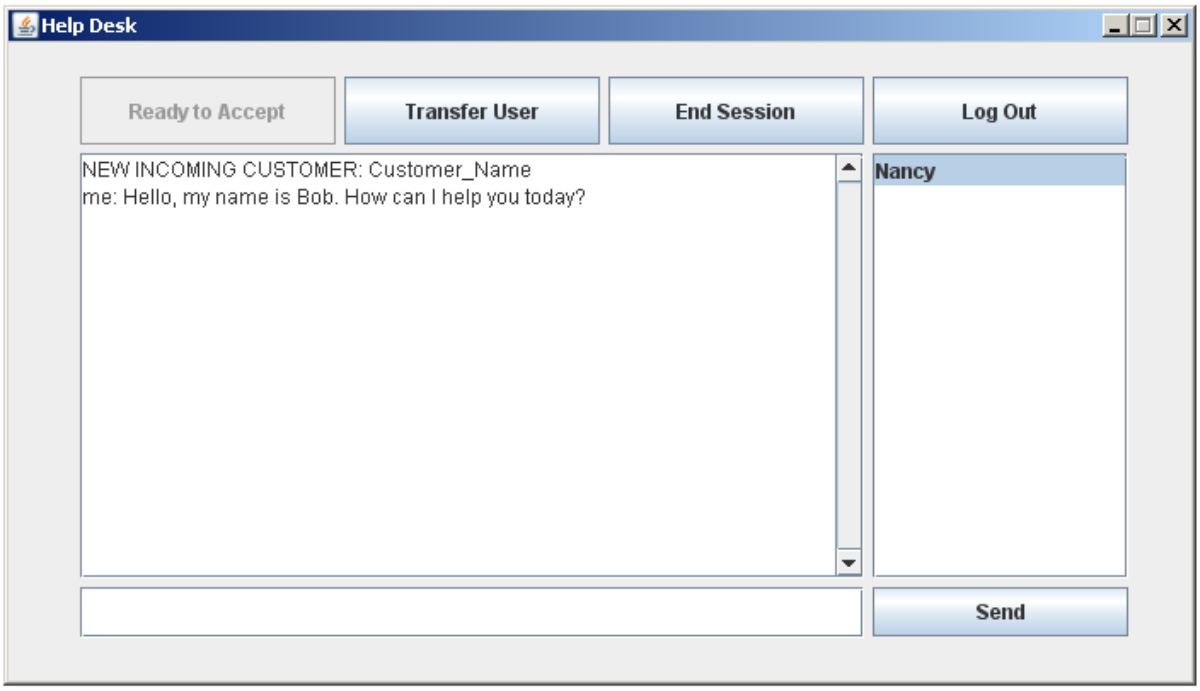
New changes from proposed design:

Simplified description. Instead of "Help Desk Authentication", "Help Desk Agent Login" sounds more standard. Also, "name" is renamed to "User Name" to fit the standard format of a login screen.

Auto-focus text field. When the help-desk login screen shows up, the text field for "User Name" is focused on.

"Back" button added. Give users the ability to go back to the Welcome screen, in the case that users want to change the connection settings or for any other purpose.

Help-Desk Chat Screen + HD to HD Chat Screen



New changes from proposed design:

Added Ready to Accept button. Help-desk users might not want to accept the customer right away, so now they are given an option of when to accept a costumer.

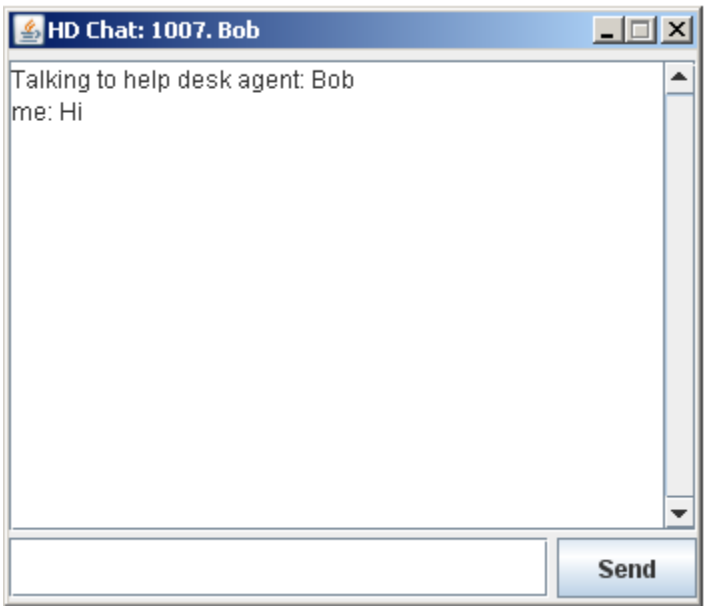
Added Send button for standard chat program format.

Moved the (Transfer, End Session, Logout) buttons to the top. This gives more space for the user list area to expand vertically.

Expanded list area. The user list is vertically stretched so that more logged on help-desk users are displayed. This gives a better view of online help-desk users.

Exit button changed to Log Out. Instead of exiting the application, it goes to the previous help-desk login screen. This provides better functionality in situations where more than one help-desk user uses the same computer. If one help-desk user is done their work shift, they can log out and leave it open for the next help-desk user to log in.

No changes to HD to HD chat screen **other than adding a send button.**



TESTING

The final phase of this application is testing. We indulged in vigorous testing by coming up with several test case scenarios and executing them.

In this section, we will describe briefly our test case scenarios and how we made sure all the requirements were satisfied.
'**' indicates a negative test case.
Consider four Help desk agents: HDUser1, HDUser2, HDUser3 and HDUser4. For the purpose of this section, assume the anonymous users go by the name anon1, anon2, anon3 and so on.

AUTHENTICATION

- In the main window, enter any name in the text box or leave it empty. You should be accepted as an anonymous user (Passed).
- In the main window, press CTRL + SHIFT + H. The screen should change to another window asking for username and password. (Passed).
- On the previous window, enter valid username and password of a help desk user. User must be authenticated and screen should change to the home page of the user.
- When the user enters invalid login information, an error message denying access to the user must be displayed (Passed) **.

A COMBINATION OF TEST CASE SCENARIOS

- HDUser1 and anon1 are logged in. HDUser1 clicks ready to accept. He automatically is assigned anon1. The two users

are able to send messages to each other. (Passed). This test case demonstrates that an HD user can send messages to an anonymous user.

- Consider the above test case. After a few messages have been exchanged, anon1 ends the conversation. The ready to accept button of the HDUser1 becomes enabled. (Passed). This test case demonstrates that an HD user can continue to be logged on even after an anonymous user leaves.

- Consider a case similar to the one above, but HDUser1 ends the conversation first. The Anonymous user must be logged out instantly.

- Five users log in this order: HDUser1, anon3, anon2, anon1, HDUser2. HDUser2 clicks ready to accept, anon3 is served. HDuser2 clicks ready to accept; anon2 is served, anon1 is still waiting to be served. (Passed). This test case shows that the order in which the anonymous users are served is based on a first-come first serve basis.

- Four users are logged in: HDUser1, anon3, anon2, HDUser2. HDUser1 is talking to anon3 and HDUser2 is talking to anon2. HDUser1 tries to transfer its anon3 to HDUser2. HDUser1 should not be allowed to do that. An error message must be displayed (Passed). This shows that an anonymous user can only be transferred to a help desk user who is not already having a conversation with another anonymous user.

- Three users are logged in: HDUser1, anon3, HDUser2. HDUser1 is talking to anon3 and HDUser2 is ready to accept. HDUser1 transfers anon3 to HDUser2. HDUser2 should now be assigned anon3.

- Four users are logged in: HDUser1, anon3, anon2, HDUser2. HDUser1 is talking to anon3 and HDUser2 is talking to anon2. HDUser1 must be able to see that HDUser2 is online and vice-versa. When HDUser1 double clicks on HDUser2, a chat window between them pops up. They are able to continuously send messages to each other. When one of them logs out, the other must be notified of that (Passed).

- Four users are logged in: HDUser1, anon3, anon2, HDUser2. Another HD User tried to login elsewhere with the credentials of HDUser2. Since, HDUser2 is already logged in, he must not be allowed to log in and error message must be displayed. (Passed).

SECURITY ISSUES

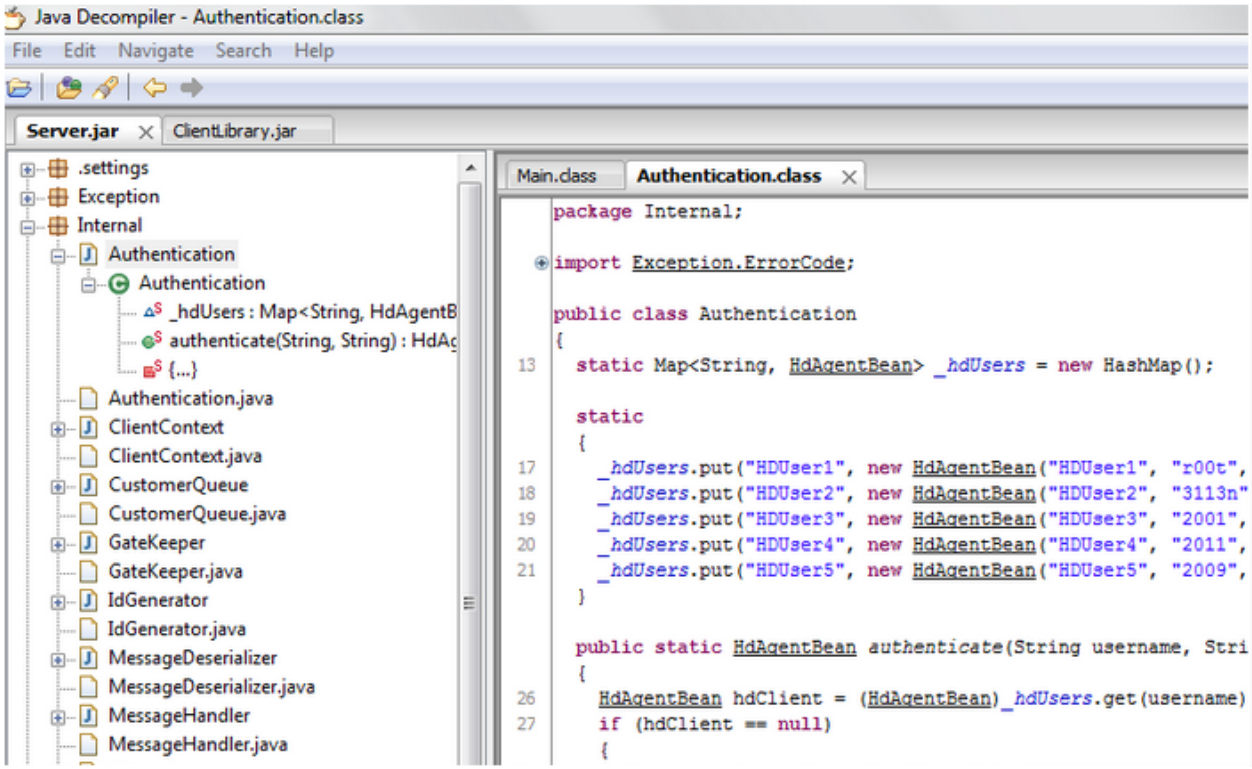
1. ISSUE #1

Attack objective: To recover passwords of Help-Desk Users.

In the current situation, it so happens that all the username – password combinations of Help-Desk Users are stored in plain text in the source code. Therefore, if we successfully gain access to the underlying source code of the application, passwords can be obtained with absolute ease.

If we use a java decompiler, we can easily recover the file that stores all the passwords.

The following is a screenshot of the class in the server project that contains all the passwords in a map.



What this tells us is that the way our server is designed right now; passwords can be easily stolen and used for malicious purposes.

We have discussed Obfuscation in issue 1. However, known De-Obfuscation techniques exist, which could be possibly used to reverse the Obfuscation process. Therefore, there is need for a stronger and more secure process; **Encryption** serves exactly that purpose.

By applying encryption to confidential information, it guarantees that the information stays secure and does not get leaked.

In phase 3, we used AES/CBC/PKCS5 Padding encryption algorithm to encrypt the records.
Password validation is being done with SHA1.
We used PBKDF2 with Hmac SHA1 to generate the key with 20,000 iteration count.

2. ISSUE #2

Attack Objective: Use a sniffing tool to collect any confidential information sent between a client and the server.

SETUP:

a) Install the Server on a machine and the Client onto another machine.

b) Start the Server.

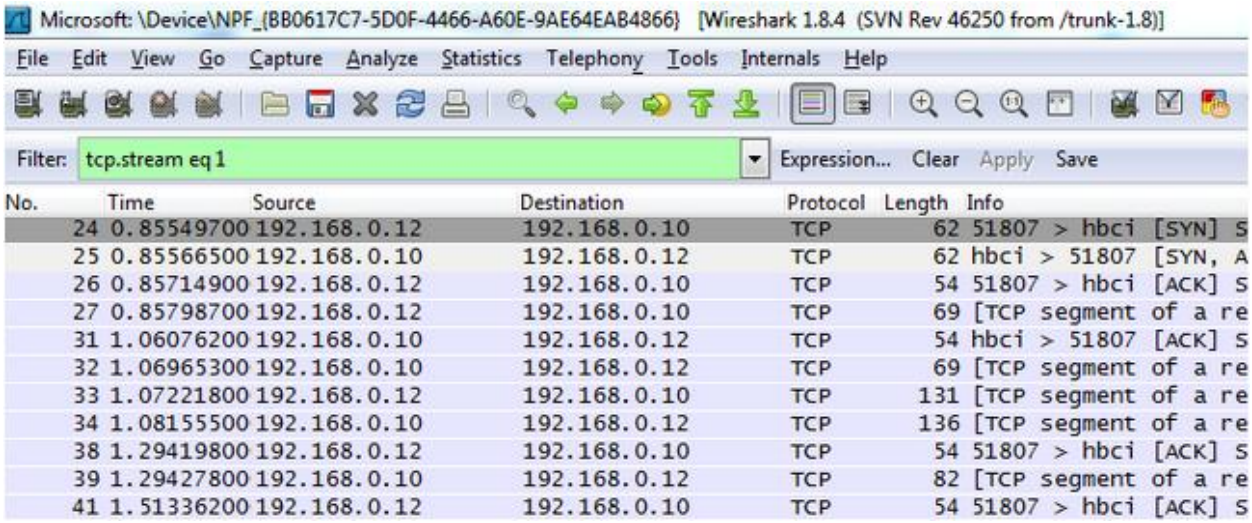
c) Run the Client.

d) Open Wireshark on the Server machine and start capturing.

e) Login as a Help-Desk user.

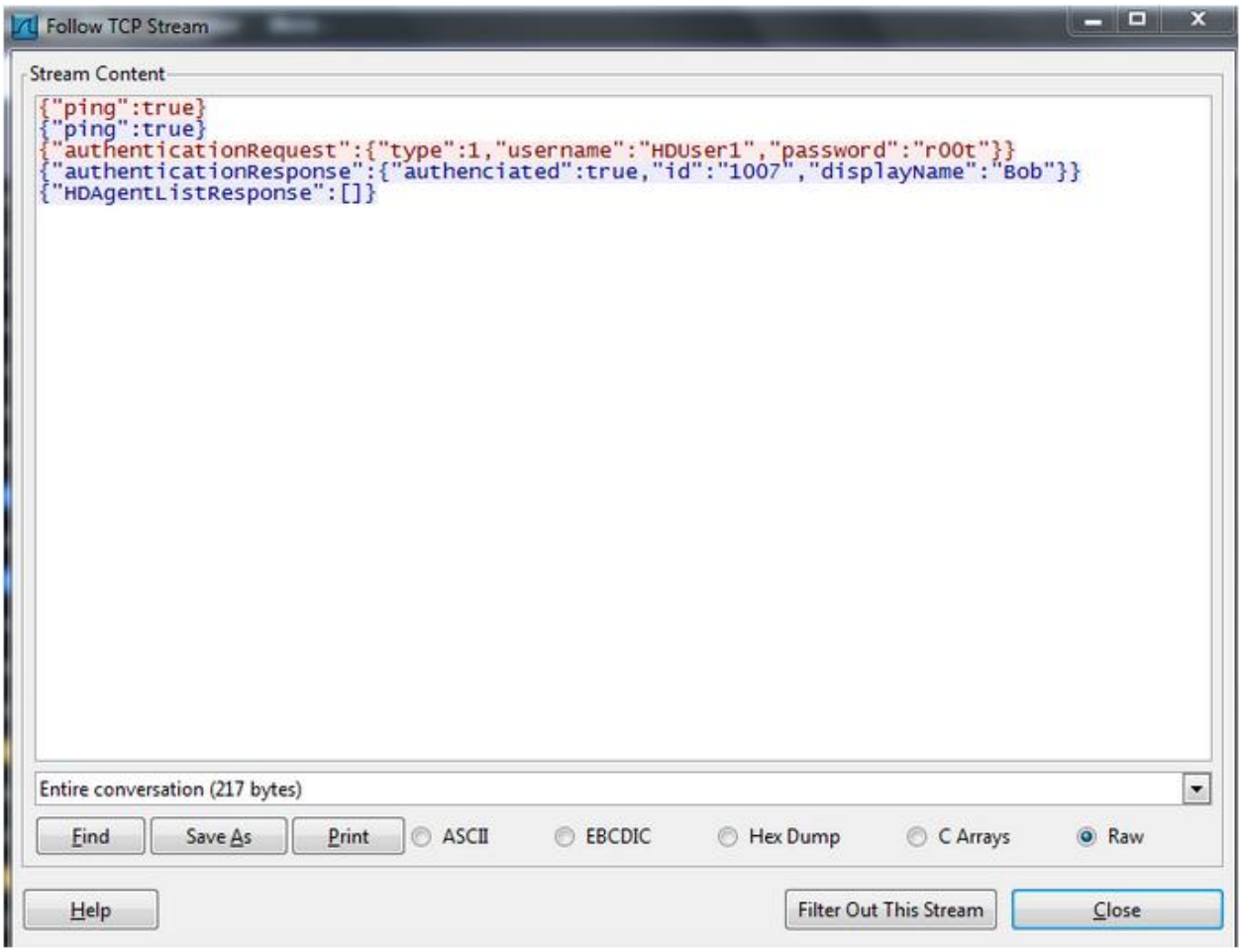
f) If Login successful, stop capturing on Wireshark.

The following screenshot shows a capture of the packets during this time



g) Follow TCP Stream on one of the packets and it shows you exactly what information was exchange between the client and server.

The following screenshot depicts that:



As you can see, information was sent in plain text. The username and password of the Help-Desk user who just logged can be seen in plain view. This is a dangerous security bug. In our current design, a malicious user can easily use a sniffing tool on the client or the server and collect confidential information without the Client's information.

Currently in our design, we use HTTP to connect the client to the server. As we know, HTTP is not secure and that is the reason the captured information on wireshark was visible in plain text. However, if the protocol to connect the client to the server was changed to secure HTTP or **HTTPS**, the information would be secured and this issue could be solved. In Phase 3, we used a TLS socket for this purpose.