



Howdy Chat Protocol and Howdy C/S Application

Jingjun Zhang, jzhang9@wpi.edu

Contents

Abstract	3
Project Description	3
User Requirements	3
Technologies and Environment	4
System Architecture	4
Detailed Design	5
Howdy Chat Protocol	5
Howdy Server	6
<i>Design Ideas</i>	6
<i>Flowchart</i>	6
<i>Data Structure</i>	8
Howdy Client	8
<i>Design Ideas</i>	8
<i>Flowchart</i>	8
<i>Data Structure</i>	9
<i>User Interface</i>	10
Test and Evaluation	10
Test Environment	10
Test Case Design	11
Test Result	11
Future Development	12
Conclusion	12
Appendices	13
References	13
Test Cases and Results	14
<i>Client GUI</i>	14
<i>Server Configuration</i>	15
<i>Single Client Connection</i>	15
<i>Single Client Username Register</i>	17
<i>Multiple Client Connection</i>	18
<i>Multiple Client Username Negotiation</i>	19

<i>Broadcast Message Exchange</i>	22
<i>Whisper Message</i>	24
<i>Texting While Receiving Message</i>	26
<i>Single User Offline</i>	27
<i>Multiple User Offline</i>	29
Source Code	30
<i>HowdyServer.py</i>	30
<i>HowdyClient.py</i>	33

Abstract

This report outlines the design and development of Howdy Chat Protocol and its client/server mode application. Howdy Chat Protocol is a very simple application layer protocol running on TCP port number 20002, designed by Jingjun Zhang, the author of this report, to achieve basic information exchange between users who want to chat with each other online. Howdy Server and Howdy Client are developed on Python 3.5.2 platform as the implementation of Howdy Chat Protocol.

Chapter 1, Project Description provides brief picture of this project to readers.

Chapter 2, Detailed Design illustrates procedure and packet format of HCP (Howdy Chat Protocol), behavior of server and client, the interactions between Howdy Server and Howdy Clients, as well as the structure of source code.

Chapter 3, Test and Evaluation describes how author designs and carries out test cases on a high performance VMware ESXi bare metal host, which runs multiple virtual machines.

Chapter 4, Future Development presents the ideas of author on how to improve this application and extend the protocol.

Chapter 5, Conclusion summarizes the process and results of this project.

Appendices provides references, detailed information on test cases and results and source code of application.

Project Description

The purpose of this project is to create a client/server system with socket to satisfy the requirement of simple online chat room.

User Requirements

The basic requirements of this system are:

- Server should be able to handle multiple clients simultaneously, including connection, disconnection, username picking and regular message exchange, without interruption of each other.
- Server should be able to display critical event while running, including connection status and user activity.
- There should be a negotiation mechanism between server and clients to decide a unique username for each online user, before a user can send or receive any message.
- There should be an update mechanism for the server to inform clients of changes of other clients' status.
- Clients should be able to choose a username before chatting.
- Clients should be able to send message to all or only whisper to a single user.
- Clients should be able to see a list of online users, including itself.
- Clients should be able to see the status change of user list.

- Clients should be able to see messages sent from each own.
- Clients should be able to see messages sent to itself or all users.
- Clients should not be able to see messages which are not supposed to be received by itself.
- Clients should be able to receive messages even when typing outgoing message.
- Clients should be able to quit the application without disrupting the server.
- Clients should have a GUI to interact with users.
- The system can be developed based on any operating system and use any popular programming language.
- Socket programming must be involved in the communication.

Technologies and Environment

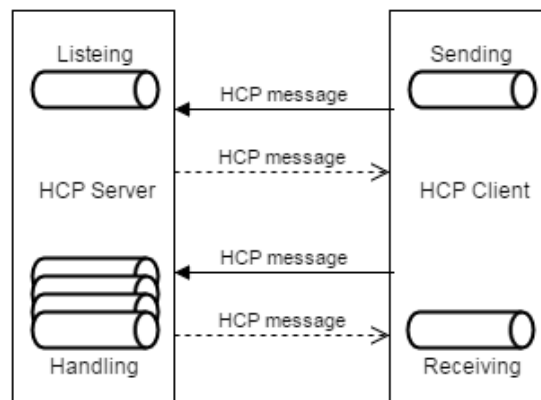
To satisfy these requirement, we need introduce:

- **Socket programming** to establish and control TCP connection as lower layer link between server and clients.
- **Multiple threads** to achieve multiple client connections on server end and separation of sending and receiving on client end.
- **A simple application layer network protocol** to divide phases of communication, negotiate username and distinguish, assemble and parse different type of messages. In this case, I designed a protocol named **Howdy Chat Protocol**.
- **A GUI development component** to provide user interface for clients.

The development is based on Python 3.5.2 platform on my MacBook Air with OS X 10.11. Under this platform, we have socket and threading as standard library to support socket programming and multiple threads. We also have tkinter as GUI development tools. PyCharm Community Edition 2016.1.4 is chosen for IDE of this project for its friendly interaction.

For testing, I ran functional test on my MacBook with VMware Fusion 7 to create a virtual machine as client, during the development period. The final system evaluation is carried out on a HPE ProLiant DL380 Gen9 server, with 16 2.6GHz CPUs, 32 logical processors. I created virtual machines as clients after installing ESXi 6.0.

System Architecture



The system is composed of a server, multiple clients and HCP message sent and received by them. The server will be running one listening thread to accept new incoming connections and multiple handling threads to handle communications with each connection. While on client side, there will be two threads, one for sending messages, and the other for receiving message. Server's socket will be set to allow reuse to accept multiple client. The detail of this system and application will be illustrated in the following chapters.

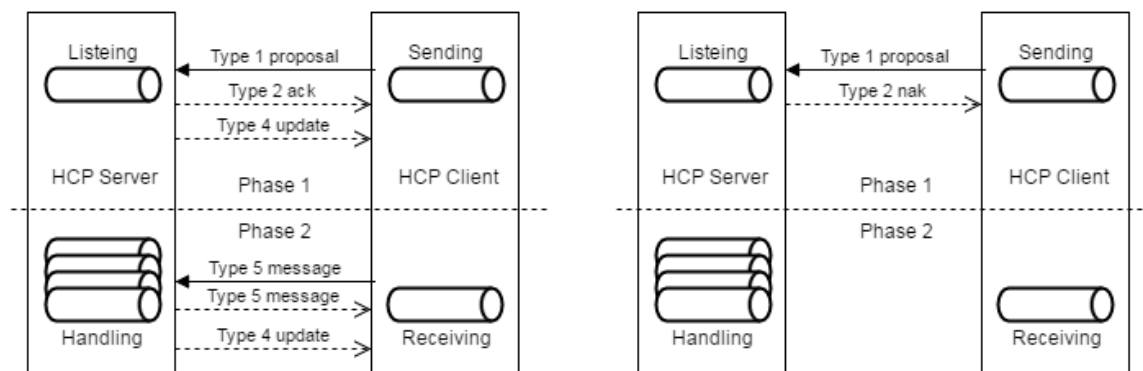
Detailed Design

We will discuss detail of HCP and software design of both server and client application here.

Howdy Chat Protocol

HCP is running on TCP port number 20002 by default, the reliability of connection can be guaranteed by TCP. The communication of HCP is divided into two phases, Phase 1 for name negotiation and Phase 2 for message exchange. And in these two phases, there will be 5 different type of messages are exchanged between server and clients. Type 1 is 'proposal' message, sent from client to server with proposed username, to inform a possible username. Type 2 is 'acknowledged' message from server with proposed username, to inform it is accepted. Type 3 is 'not acknowledged' message from server to inform the proposed username is rejected. Type 4 is 'update' message from server to inform the newest user list on server, and it is triggered by event, like user logon and logoff. Type 5 is 'message' message, it could be send either from server to client or from client to server. Type 5 message carries text of conversation between users.

In phase 1, client will send Type 1 message to server. If the proposed username is accepted, server will send back a Type 2 message to confirm the username then update client with current user list, which means the end of Phase 1 and start of Phase 2, then online users can talk with each other. Otherwise, server will send a Type 3 message to reject the username explicitly, which means HCP stays in Phase 1.



In Phase 2, clients can send Type 5 message to server and server will forward Type 5 message to all other clients or one specific client according the receiver of message. Whenever there is an update of online user list, server will send Type 4 message to all online users to refresh user list on clients.

Message type is represented with 1 bytes length, and valid value will be 0x01 to 0x05. Length of username in HCP message is 64-bytes. For message Type 1 to 3, the format of message should be

message type followed by proposed username. For message Type 4, the format message should be message type followed by a list of usernames, each of which will be 64-bytes long. For message Type 5, the format of message should be message type followed by username of sender, username of receiver and original message. 'All' and 'Server' is reserved as username for all the online user, so no one can chose them as usernames.

The following picture is the format of HCP messages:

Type 1 proposal	message type (1 byte)	proposed username (64 bytes)	
Type 2 ack	message type (1 byte)	proposed username (64 bytes)	
Type 3 nak	message type (1 byte)	proposed username (64 bytes)	
Type 4 update	message type (1 byte)	online user 1 (64 bytes)	online user 2 (64 bytes)
Type 5 message	message type (1 byte)	sender username (64 bytes)	receiver username (64 bytes) message (variable length)

There is no consideration for disconnection since HCP can rely on TCP to handle it.

Howdy Server

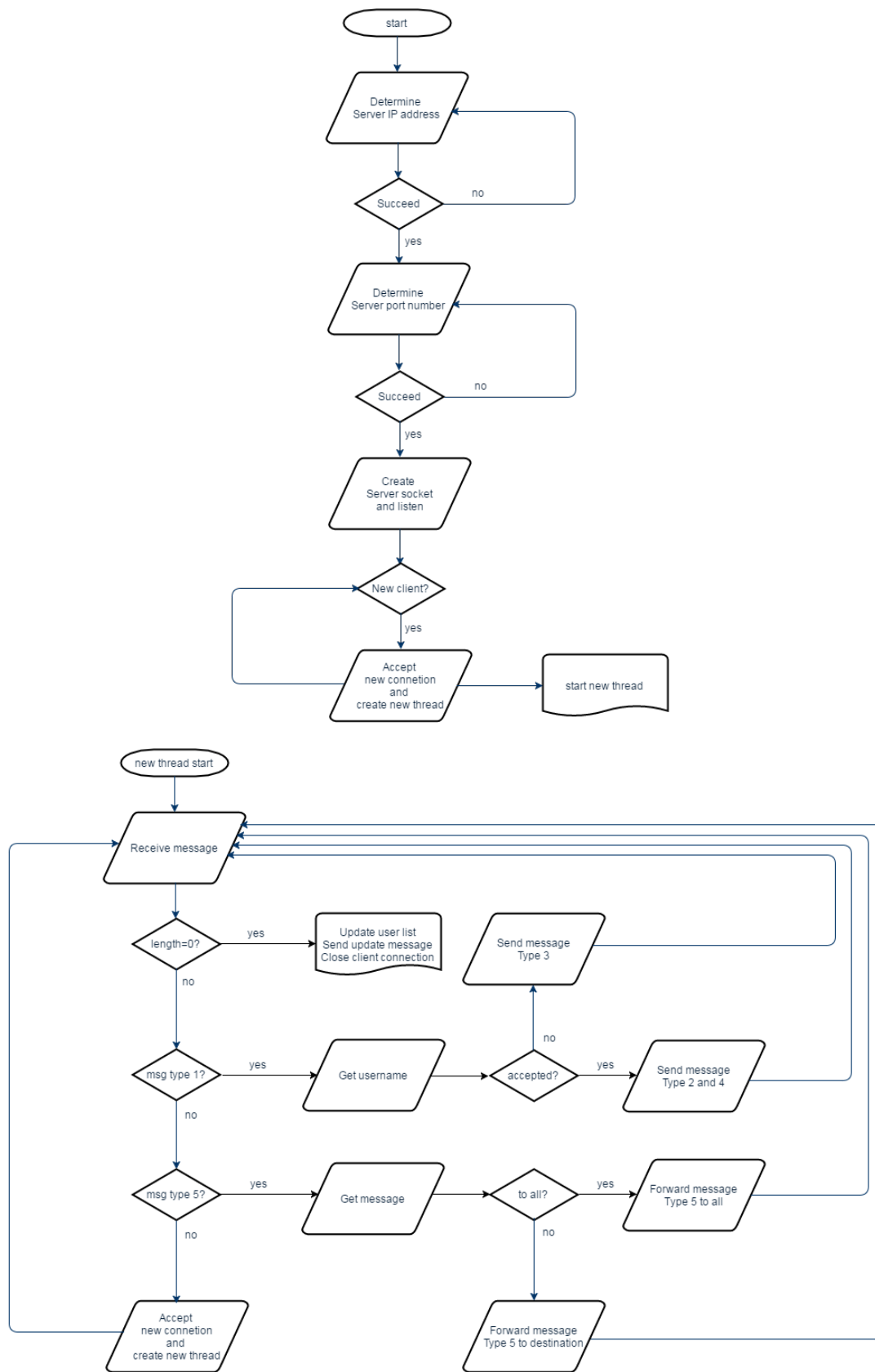
Design Ideas

The major ideas about design of Howdy server includes:

- Keep server socket opening to listen new incoming client connections. Allow server socket to reuse IP address and port number of server.
- For each new incoming client connection, create a new thread to receive HCP message from this client, handle or forward messages according type and receiver of HCP messages, and deal with username negotiation and change of connection status.
- Maintain a dictionary for mapping between online username and client connection, and make it accessible for each thread so that the threads can modify the dictionary once there is any change on connection status.

Flowchart

The flowchart of original listening thread and threads created for new client connection is shown as below. When the server is started, there will be a command line prompt asking for IP address and port number to run the HCP service. The Howdy server will use them to start a new socket and listen for new clients. When a client connects server, a new thread will be created to handle this client connection, and the server socket will keep listening for next client. For the new thread of client connection, it will check the connectivity of client socket. If connectivity is lost, it will refresh user list, inform all remaining clients and close socket. Otherwise it will check message received, see if the message is type 1 or 5, BTW server will discard messages other than type 1 or 5 silently. For message type 1, server will fetch username and check its availability and inform users. For message type 5, server will forward the message to proper destination.



Data Structure

A new class named Chatconnection is defined for dealing with client connections individually. This class inherit from threading.Thread class and is modified according to the requirement of Howdy Server application. The first change is a class variable named clients is added, whose type is dictionary, to record current users and their corresponding sockets. The second change is I re-write run method of threading.Thread so that it can deal with receiving data as long as I start a instance of Chatconnection.

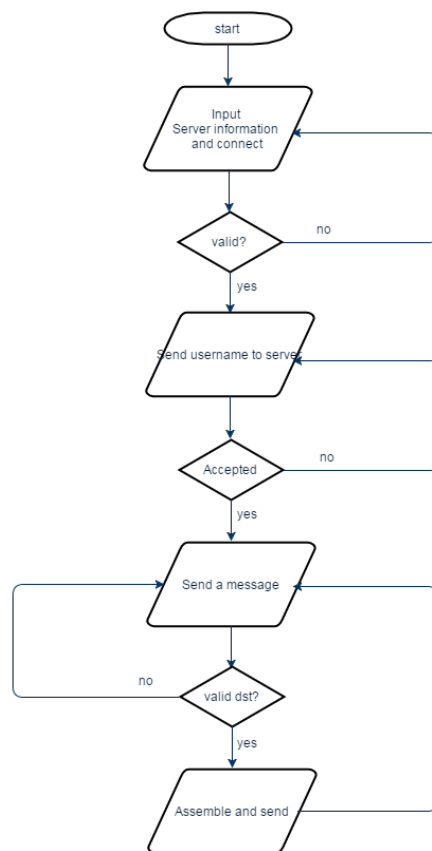
Howdy Client

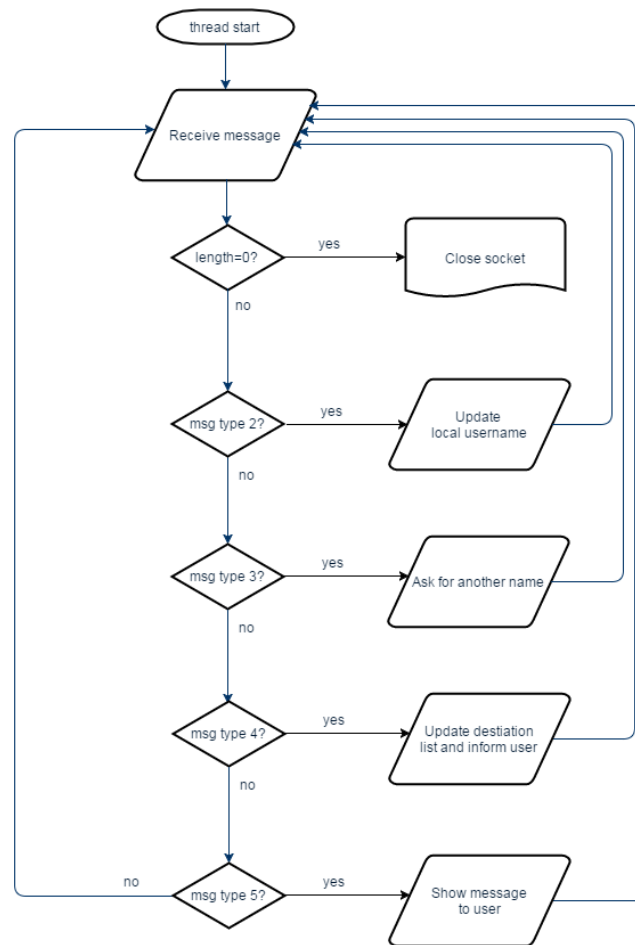
Design Ideas

The major ideas about design of Howdy server includes:

- GUI will be launched as soon as the Howdy Client starts, and future message sending will use the same thread with GUI. To send any message, user have to specify a destination from user list.
- User should send name to server but cannot start a new thread for receiving message until server send HCP type 2 message.
- The receiving thread will only deal with received data and decide how to present them to user.

Flowchart





The flowchart on Howdy Client application is shown as above. There are two flowcharts for sending and receiving thread. Packet sending share a thread with GUI, and it get message from GUI. First of all, it has to get IP address and port number from user input, and will not proceed until it connects the Howdy Server successfully. The next step is to provide a valid username to negotiate with server, and no message can be sent or received until the username is set. Message to be sent will also be checked and it will not be sent until it is valid. For receiving thread, it will check the connectivity of sock first, and will close the socket if it is not available anymore. Then it will check the message type, it can only take message type 2, 3, 4 and 5. Howdy Client will ignore message type 1 and discard unrecognized packets silently. If it received a message type 2, it will update its own username; if it receives a message type 3, it will inform user to pick a new name; if it receives a message type 4, it will update local user list and compare it with older one to find who went online or offline.

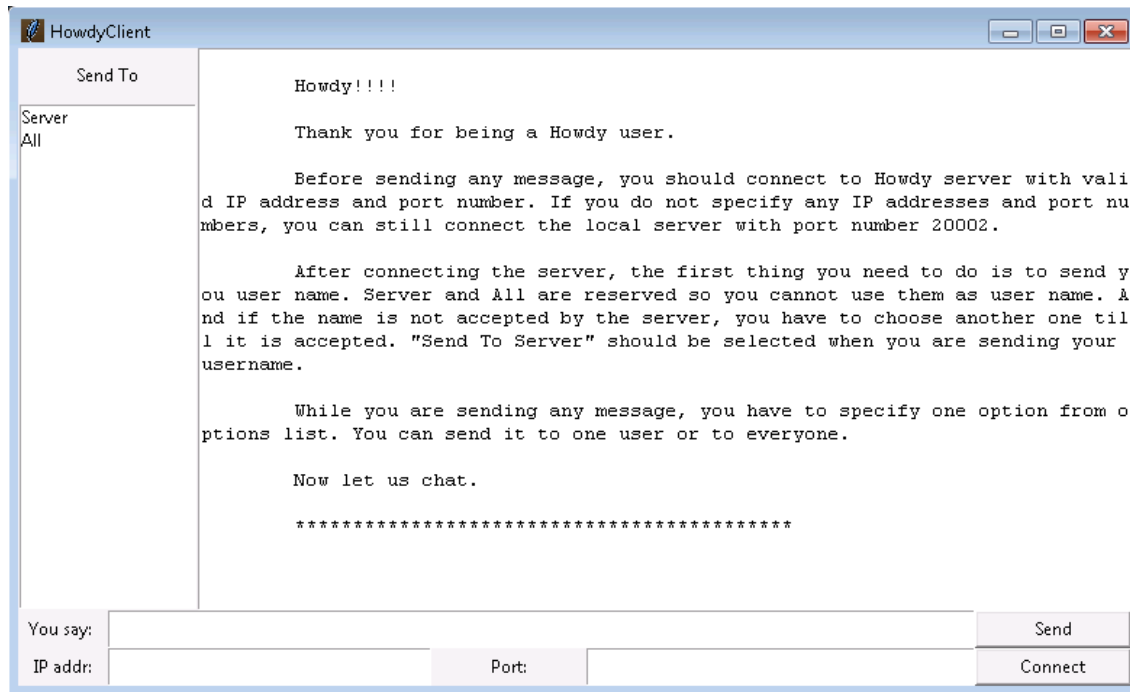
Data Structure

A new class named MultiSockClient is created for Howdy client. This client include a pre-defined GUI widget-set and layout so whenever the instance of this class is created, the GUI will be running. This class includes 5 methods: .winclose() is used to clean socket and GUI when the client is shutdown; .connectsrv() is used to create and connect socket with given parameters, after connect button is clicked; .msgsend() is triggered once user click send button or hit 'enter' key in

message entry, it will assemble message according to the destination user selected and send it out; .msgrecv() is trigger after connect server successfully, and it will parse message and display it to user; .optupdate() is used to replace items in list box with newly received user list.

User Interface

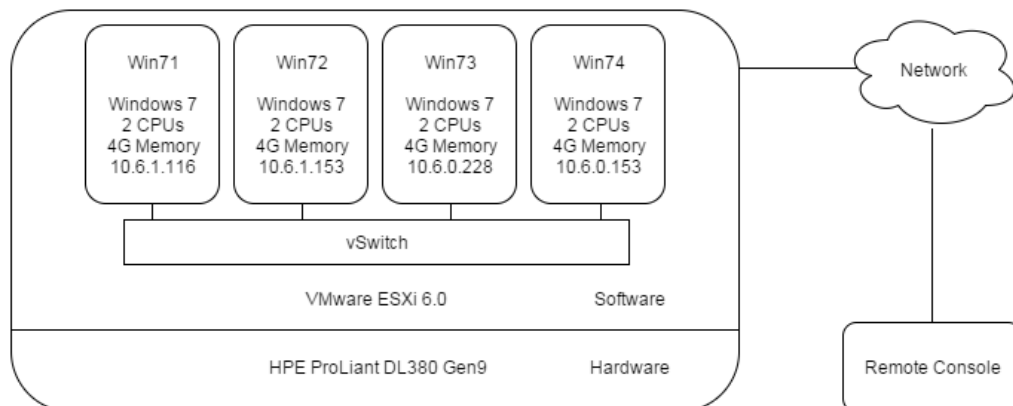
The following is the GUI design of Howdy Client.



Test and Evaluation

Test Environment

System test is running on a remote HPE Server, the environment is shown as below.



4 Windows 7 guest machines are created on ESXi bare-metal host, and each of them are allocated with 2 CPUs and 4G memory. The management of virtual machines is based on vSphere Client and Remote Desktops.

Win71 is running as server and the rest of them is running as clients.

Test Case Design

Design of test cases are based on user requirement of this project and all the requirements are verified in these test case. The following two lists are key requirement and how test cases cover these key requirement.

1. Multiple client.
2. Clients must be able to “whisper” to each other.
3. Clients must be able to choose a nickname.
4. Server operations should be printed out by the server.
5. The server must handle connections/disconnections without disruption of other services.
6. Clients must have unique nicknames, duplicates must be resolved before allowing a client to be connected.
7. All clients must be informed of changes in the list of connected users.
8. A list of online users must be displayed.
9. Connection/disconnection actions of users must be displayed on clients.
10. The messages you send must also be displayed.
11. Must still be able to receive messages/actions while typing a message.
12. Clients must be able to disconnect without disrupting the server.

Test case coverage:

- Client GUI, covers requirement 3, 8;
- Server Configuration, covers requirement 4;
- Single Client Connection, covers requirement 3, 4;
- Single Client Username Register, covers requirement 3, 4, 7, 8, 9;
- Multiple Client Connection, covers requirement 1, 4, 5;
- Multiple Client Username Negotiation, covers requirement 1, 4, 5, 6, 7, 8, 9;
- Broadcast Message Exchange, covers requirement 1, 4, 8, 10;
- Whisper Message, covers requirement 1, 2, 4, 8, 10;
- Texting While Receiving Message, covers requirement 2, 4, 8, 10, 11;
- Single User Offline, covers requirement 1, 2, 4, 5, 7, 8, 9, 10, 12;
- Multiple Users Offline, covers requirement 1, 2, 4, 5, 7, 8, 9, 10, 12.

Test Result

Howdy Server and Client passed all test cases including some exception test not included in this report. This product shows great quality, availability and accessibility. For detailed test result, please refer appendices.

Future Development

Current system is the best product I can offer under the limitation of development period, my knowledge base and requirement of project. There are still further improvements that can be implemented on this system, and I will consider following possibilities:

- Replace socket with socketserver library on server side, since socketserver is more mature and graceful solution on Python platform for TCP multi-client server.
- Implement a user database on server side, so server can introduce user credential and save them into database.
- Introduce server log system to record running event.
- Introduce user roles into this system, so administrator can manage server remotely.
- Introduce user profile into the system, so users can get more information about the person who is talking.
- Implement creation of exclusive club and enable sending message within this exclusive club.
- Enable exchange of message besides text, for example, images or videos.
- Introduce TLV structure in HCP to support variable length and arbitrary combination of values in the same HCP message.

Conclusion

This report demonstrated a successful socket programming project with client and server architecture and a customized application layer protocol to make sure the interaction between client and server under control. For network programming project, the understanding and design of interaction between network entities is critical, and it also help me clear the structure of software components and source code. The design and development of this system considered most possible exceptions and handle them well. Howdy server and client application satisfied all the requirements and showed us a stable software quality by passing all test cases.

Appendices

References

Python Standard Library: <https://docs.python.org/3/library/>

Tkinter 8.5 reference: a GUI for Python:

<http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html>

Python 3 Tutorial by Xuefeng Liao:

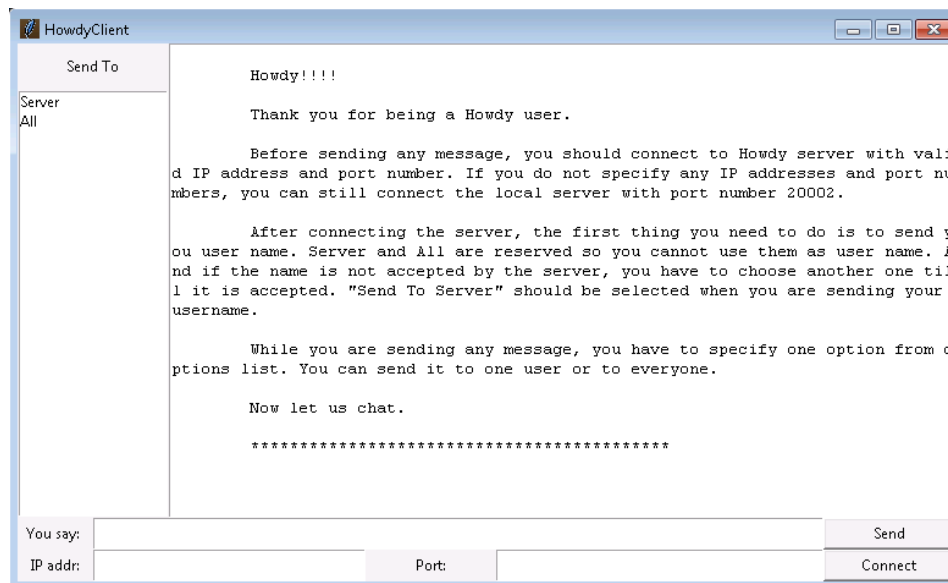
<http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000>

Test Cases and Results

Client GUI

Test Case Description			
This case is designed to validate the elements in GUI of client, to make sure the GUI can provide user input and output support the functional features of Howdy Client.			
Pre-Conditions			
Start Howdy Client in Python IDLE on host Win72			
Step	Actions	Expected Output	Success/Fail
1	Check window widgets for connecting servers.	There should be text entries to specify server information and a button to connect.	Success
2	Check window widgets for input username.	There should be an entry to specify username and send	Success
3	Check window widgets for text message sending.	There should be an entry to input message and send.	Success
4	Check window widgets for text message receiving.	There should be a text box to display receiving messages.	Success
5	Check window widgets for specifying user as message destination.	There should be a list for user selection.	Success
6	Check window widgets for displaying user list.	There should be a list box to display user list.	Success
Test Result			
Test result is success. Howdy Client provide a GUI with proper input and output method for user to interact with server. It creatively combine the input of Username and Message into one text entry, and also combine the function of displaying user list and specifying user as destination in a single list box. It makes the GUI more compact and user friendly. Howdy Client GUI is shown below.			

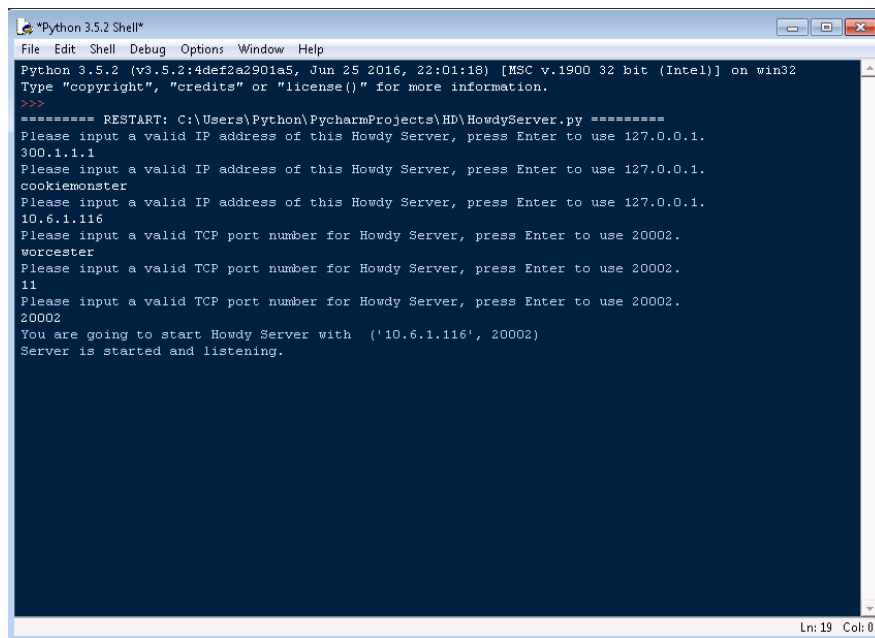
Howdy Client GUI:



Server Configuration

Test Case Description			
This case is designed to validate the configuration of Howdy Server, checking if server can take user manual setting value and run.			
Pre-Conditions			
Start Howdy Client in Python IDLE on host Win71			
Step	Actions	Expected Output	Success/Fail
1	Input an invalid IP address or random string while being asked for IP address.	Server will ask for a valid IP address again.	Success
2	Input a valid IP address.	Server will ask for port num.	Success
3	Input an invalid TCP port number or reserved TCP port number.	Server will ask for a valid port number again.	Success
4	Input a valid port number.	Server will start running.	Success
Test Result			
<p>Test result is success.</p> <p>Howdy Server can distinguish valid and invalid IP address/TCP port number, and take these valid values as running parameters. Server configuration is shown as below.</p>			

Server Configuration:



```

Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Python\PycharmProjects\HD\HowdyServer.py =====
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
300.1.1.1
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
cookiemonster
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
10.6.1.116
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
worcester
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
11
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
20002
You are going to start Howdy Server with ('10.6.1.116', 20002)
Server is started and listening.
Ln: 19 Col: 0

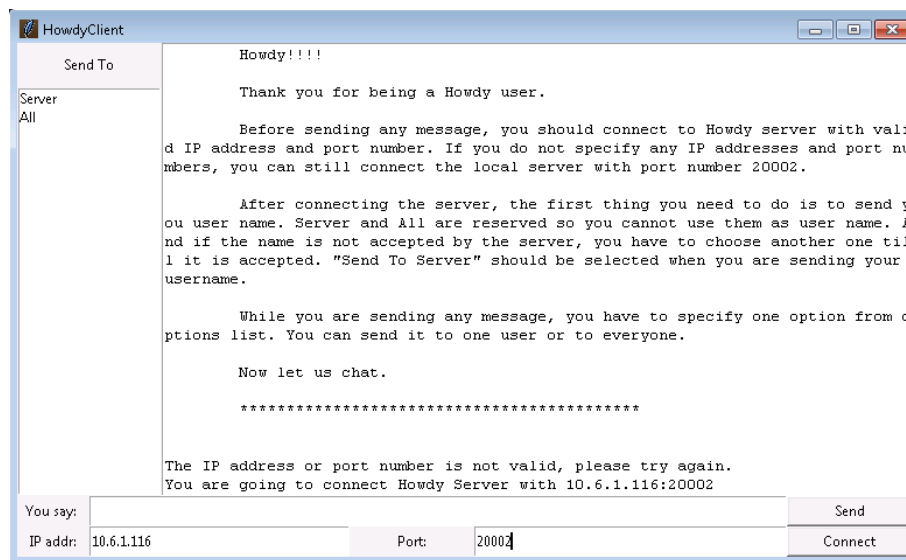
```

Single Client Connection

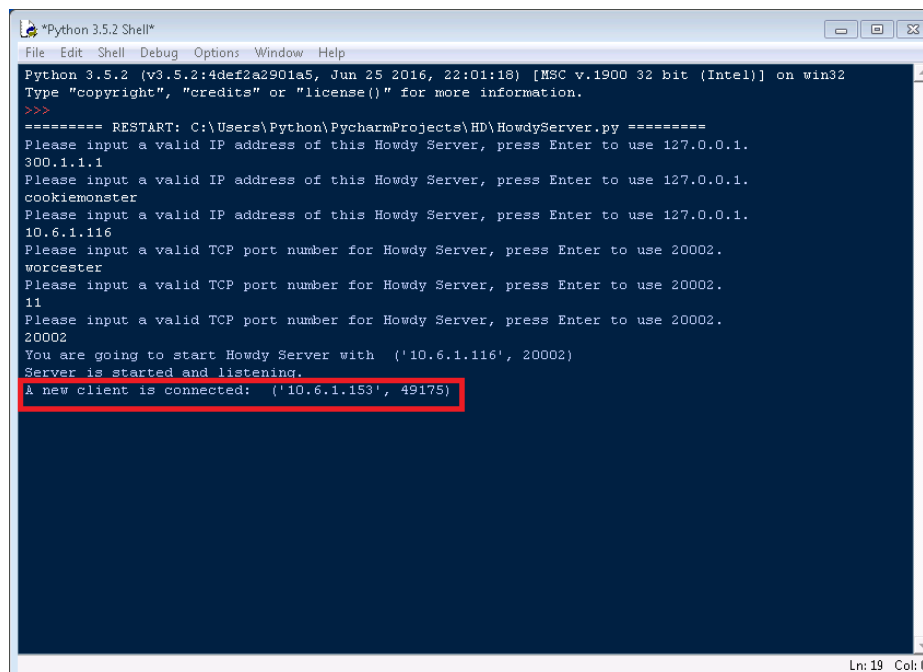
Test Case Description			
This case is designed to validate connection between server and a single client.			
Pre-Conditions			
A server and a client are started in Python IDLE of two virtual machines respectively. The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and the client is running on host Win72.			
Step	Actions	Expected Output	Success/Fail
1	Connect with invalid input of IP address and port number.	Client will print error message in text box.	Success

2	Connect with valid input of IP address and port number.	Client will connect server successfully. Server will display the event of a new connection.	Success
Test Result			
<p>Test result is success.</p> <p>Howdy client can connect server only with valid parameters and server can show event of new connection. Test result is shown as below.</p>			

Single Client Connection:



Server Event:



Single Client Username Register

Test Case Description			
This case is designed to validate username selection of a single client.			
Pre-Conditions			
The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and the client is running on host Win72. The client has connected to the server.			
Step	Actions	Expected Output	Success/Fail
1	Select 'All' and send message 'Hi, my name is Jon Stewart.'	Client will not send message and will tell user to get a name.	Success
2	Input 'Jon Stewart' and select 'Server' as destination, then send message.	Server will display event of a new username is received and accepted, as well as receiving of type 1 message and sending of message type 2 and type 4. Client will display the username is accepted and user list is updated.	Success
Test Result			
<p>Test result is success.</p> <p>Client can send username to server when target is selected as server. Server shows events of receiving Type1, checking and accepting of username and sending of Type2 and 4. Clients update the user list. The test results are shown as below.</p>			

Single Client Username Register:

HowdyClient

Send To: Server, All, Jon Stewart

Before sending any message, you should connect to Howdy server with valid IP address and port number. If you do not specify any IP addresses and port numbers, you can still connect the local server with port number 20002.

After connecting the server, the first thing you need to do is to send your user name. Server and All are reserved so you cannot use them as user name. And if the name is not accepted by the server, you have to choose another one till it is accepted. "Send To Server" should be selected when you are sending your username.

While you are sending any message, you have to specify one option from options list. You can send it to one user or to everyone.

Now let us chat.

The IP address or port number is not valid, please try again.
 You are going to connect Howdy Server with 10.6.1.116:20002
 You need set a username first.
 Your username is accepted
 Jon Stewart
 join this conversation.

You say: Send

IP addr: Port: Connect

Events on Server:

```

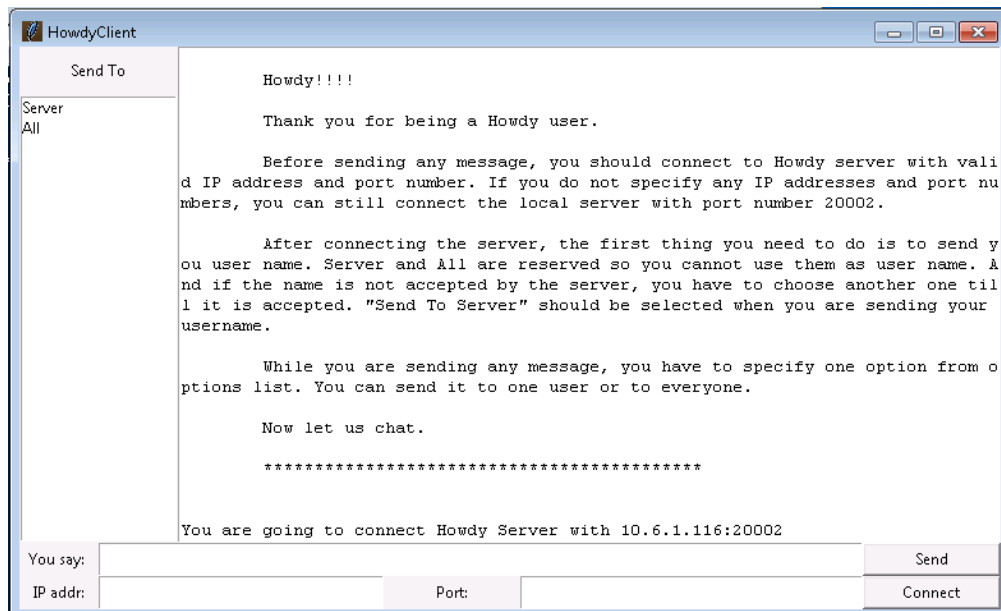
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Python\PycharmProjects\HD\HowdyServer.py =====
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
300.1.1.1
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
cookiemonster
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
10.6.1.116
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
worcester
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
11
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
20002
You are going to start Howdy Server with ('10.6.1.116', 20002)
Server is started and listening.
A new client is connected: ('10.6.1.153', 49175)
New message received. Message type: 1
Parsing packet type 1 from ('10.6.1.153', 49175)
A username is received: Jon Stewart
A username is accepted: Jon Stewart
Accepted user comes from ('10.6.1.153', 49175)
New user is added into list. Current User list: ('Jon Stewart': <socket.socket fd=388, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('10.6.1.116', 20002), raddr=('10.6.1.153', 49175)>)
Sending packet type 2 to ('10.6.1.153', 49175)
Sending packet type 4 to all receivers
Ln: 19 Col: 0

```

Multiple Client Connection

Test Case Description			
This case is designed to show how Howdy Server and Howdy Client collaborate to deal with multiple user connecting server.			
Pre-Conditions			
The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and the client is running on host Win72. Win72 has connected to the server. Then start client on Win73, Win74.			
Step	Actions	Expected Output	Success/Fail
1	Input server IP address and port number then click connect on Win73.	Server shows a new connection with IP address of Win73 is established.	Success
2	Input server IP address and port number then click connect on Win74.	Server shows a new connection with IP address of Win74 is established.	Success
Test Result			
Test result is success. Both two new clients connect server successfully and server print these event. The test results are shown below.			

Multiple Clients Connection (Win73 and Win74):



Events on Server:

```

Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\Python\PycharmProjects\HD\HowdyServer.py =====
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
300.1.1.1
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
cookiemonster
Please input a valid IP address of this Howdy Server, press Enter to use 127.0.0.1.
10.6.1.116
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
worchester
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
11
Please input a valid TCP port number for Howdy Server, press Enter to use 20002.
20002
You are going to start Howdy Server with ('10.6.1.116', 20002)
Server is started and listening.
A new client is connected: ('10.6.1.153', 49175)
New message received. Message type: 1
Parsing packet type 1 from ('10.6.1.153', 49175)
A username is received: Jon Stewart
A username is accepted: Jon Stewart
Accepted user comes from ('10.6.1.153', 49175)
New user is added into list. Current User list: ('Jon Stewart': <socket.socket id=388, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('10.6.1.116', 20002), raddr=('10.6.1.153', 49175)>)
Sending packet type 2 to ('10.6.1.153', 49175)
Sending packet type 4 to all receivers
A new client is connected: ('10.6.0.228', 49172)
A new client is connected: ('10.6.0.153', 49198)
Ln: 19 Col: 0

```

Multiple Client Username Negotiation

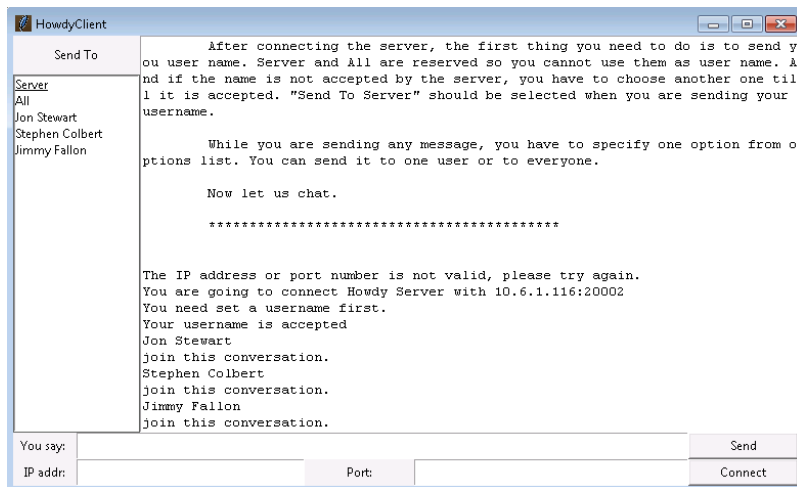
Test Case Description

This case is designed to show how Howdy Server and Howdy Client collaborate to deal with multiple user name negotiation.

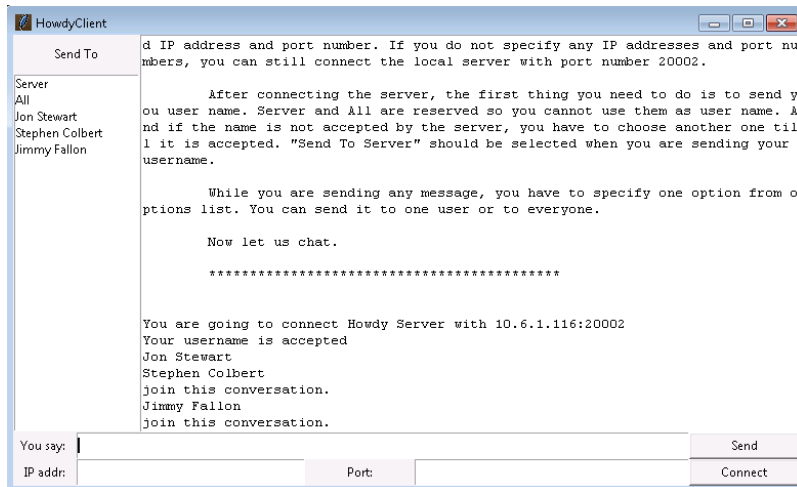
Pre-Conditions

The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and all 3 clients are running and have connected to Win71.			
Step	Actions	Expected Output	Success/Fail
1	On Win73, user input 'Stephen Colbert' as user name and send it to 'Server'.	Server will display event of a new username 'Stephen Colbert' is received and accepted, as well as receiving of type 1 message and sending of message type 2 and type 4. Win72 will display updated user list and new user join the conversation. Win73 will display that name is accepted, and user list is updated.	Success
2	On Win74, user input 'Stephen Colbert' as user name and send it to 'Server'.	Server will display event of a new username 'Stephen Colbert' is received but no other events. Nothing will happen on Win72 or Win73. Win74 will be informed that the username is not accepted.	Success
3	On Win74, user input 'Jimmy Fallon' as user name and send it to 'Server'.	Server will display event of a new username 'Jimmy Fallon' is received and accepted, as well as receiving of type 1 message and sending of message type 2 and type 4. Win72 and Win73 will display updated user list and new user join the conversation. Win74 will display that name is accepted, and user list is updated.	Success
Test Result			
<p>Test result is success.</p> <p>Server can check proposed username with current user list, accept or reject the username and shows events of receiving Type1, checking and accepting of username and sending of Type2 and Type4. The user list on each clients can be update properly when new user join the conversation. The test results are shown below.</p>			

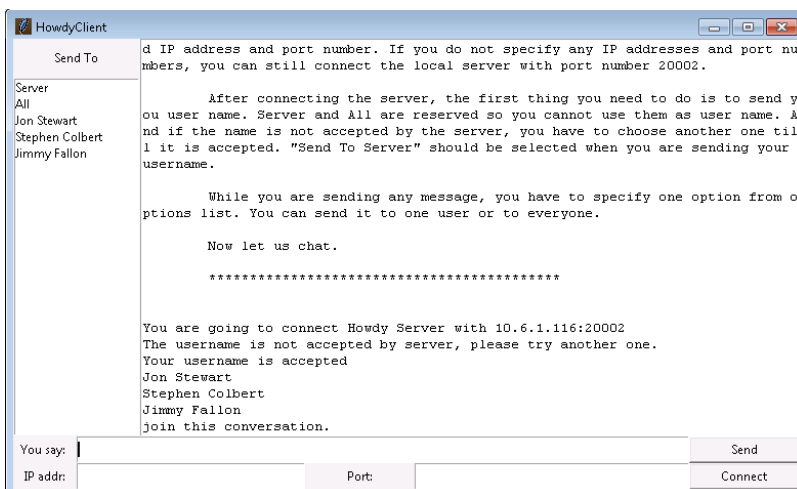
Multiple Clients Username Negotiation on Win72



Multiple Clients Username Negotiation on Win73



Multiple Clients Username Negotiation on Win74



Multiple Clients Username Negotiation on Server

```

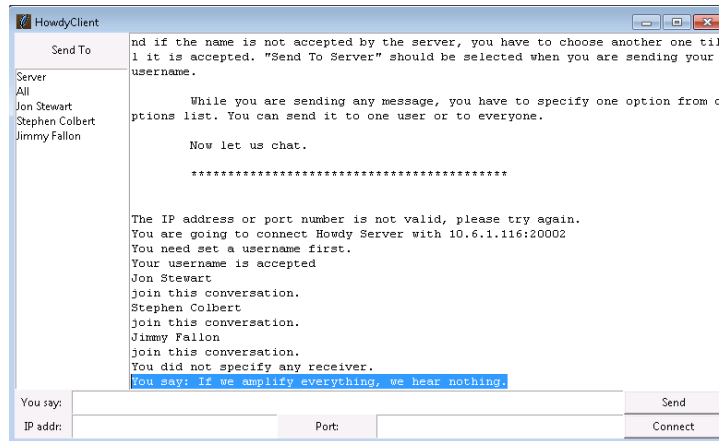
Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
>>> 234.0.0.0
Sending packet type 2 to ('10.6.1.153', 49175)
Sending packet type 4 to all receivers
A new client is connected: ('10.6.0.228', 49172)
A new client is connected: ('10.6.0.153', 49198)
New message received. Message type: 1
Parsing packet type 1 from ('10.6.0.228', 49172)
A username is received: Stephen Colbert
A username is accepted: Stephen Colbert
Accepted user comes from ('10.6.0.228', 49172)
New user is added into list. Current User list: {'Jon Stewart': <socket.socket fd=388, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('10.6.1.116', 20002), raddr=('10.6.1.153', 49175)>, 'Stephen Colbert': <socket.socket fd=8, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('10.6.1.116', 20002), raddr=('10.6.0.228', 49172)>}
Sending packet type 2 to ('10.6.0.228', 49172)
Sending packet type 4 to all receivers
New message received. Message type: 1
Parsing packet type 1 from ('10.6.0.153', 49198)
A username is received: Stephen Colbert
Sending packet type 3 to ('10.6.0.153', 49198)
New message received. Message type: 1
Parsing packet type 1 from ('10.6.0.153', 49198)
A username is received: Jimmy Fallon
A username is accepted: Jimmy Fallon
Accepted user comes from ('10.6.0.153', 49198)
New user is added into list. Current User list: {'Jon Stewart': <socket.socket fd=388, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('10.6.1.116', 20002), raddr=('10.6.1.153', 49175)>, 'Stephen Colbert': <socket.socket fd=8, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('10.6.1.116', 20002), raddr=('10.6.0.228', 49172)>, 'Jimmy Fallon': <socket.socket fd=392, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('10.6.1.116', 20002), raddr=('10.6.0.153', 49198)>}
Sending packet type 2 to ('10.6.0.153', 49198)
Sending packet type 4 to all receivers
Ln: 19 Col: 0

```

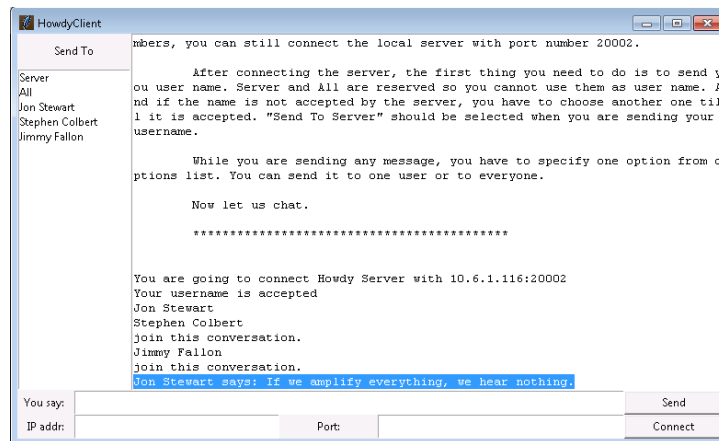
Broadcast Message Exchange

Test Case Description			
This case is designed to verify the sending and receiving of broadcast message in Howdy system.			
Pre-Conditions			
The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and all 3 clients are running and have connected to Win71 with unique username.			
Step	Actions	Expected Output	Success/Fail
1	On Win72, input 'If we amplify everything, we hear nothing.' Select 'All' and click send button.	On Win72, its own words will be displayed as 'You say: If we amplify everything, we hear nothing.' On server, there will be an event of receiving and forwarding message types and sender/receiver will be displayed. On other clients, there will be a message displayed on screen. 'Jon Stewart says: If we amplify everything, we hear nothing.'	Success
Test Result			
Test result is success. All other clients received the message and local clients show its own message. Server print relevant event. The test results are shown below.			

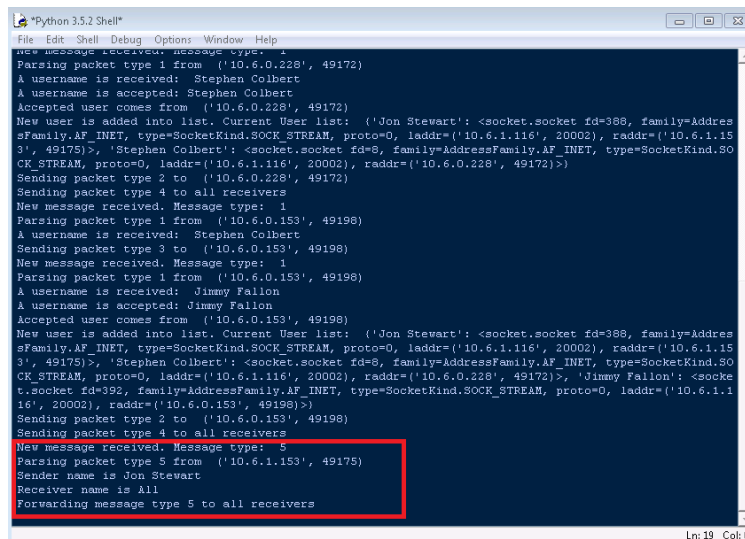
Broadcast Message on Win72



Broadcast Message on Win73/Win74



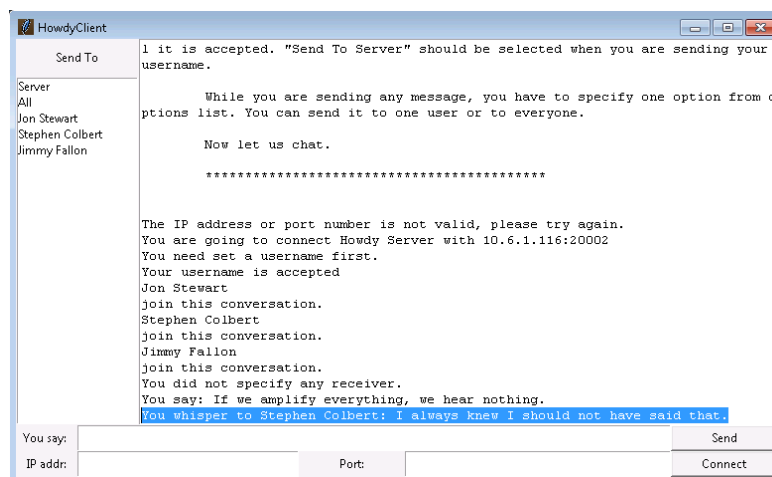
Server Event of Broadcast Message



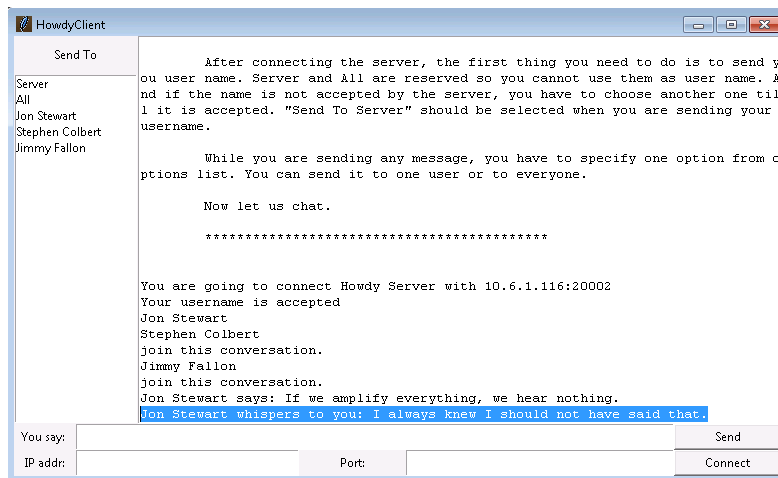
Whisper Message

Test Case Description			
This case is designed to verify the sending and receiving of unicast message in Howdy system.			
Pre-Conditions			
The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and all 3 clients are running and have connected to Win71 with unique username.			
Step	Actions	Expected Output	Success/Fail
1	On Win72, input 'I always knew I should not have said that.' Select 'Stephen Colbert' and click send button.	On Win72, its own words will be displayed as 'You whisper to Stephen Colbert: I always knew I should not have said that.' On server, there will be an event of receiving and forwarding message type5, sender is Jon Stewart and receiver is Stephen Colbert. On Win73, there will be a message displayed on screen. 'Jon Stewart whispers to you: I always knew I should not have said that.' On Win74, nothing is displayed.	Success
Test Result			
<p>Test result is success.</p> <p>Target clients received the message marked as whispered and local clients show its own message marked as whispered as well. Server print relevant event. No other clients will be disturbed.</p> <p>The test results are shown below.</p>			

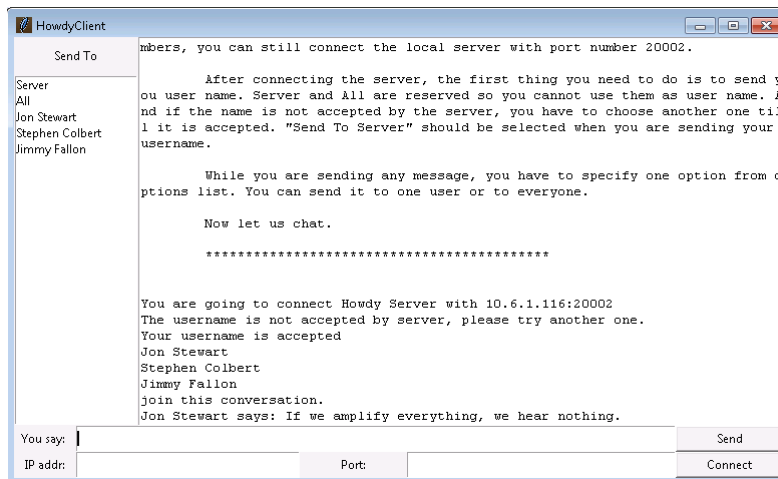
Whisper Message on Win72



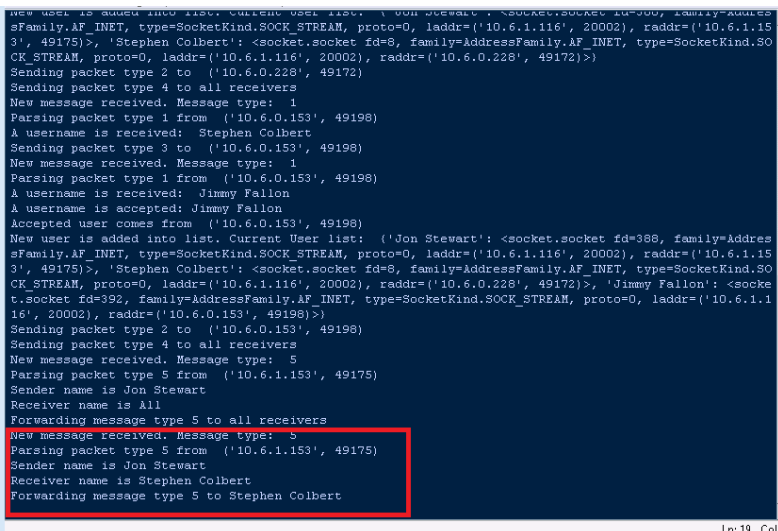
Whisper Message on Win73



Whisper Message on Win74



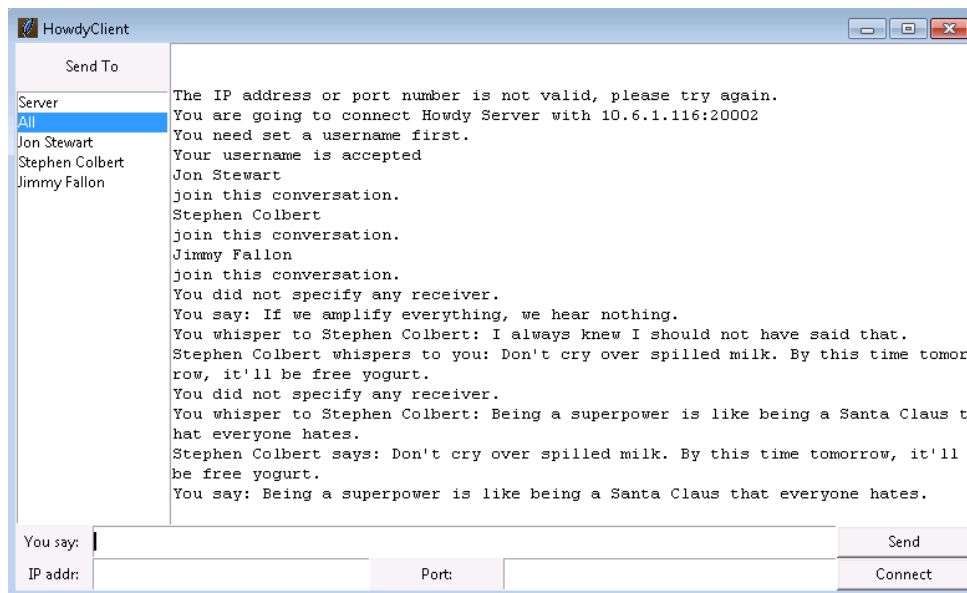
Server Events of Whisper Message



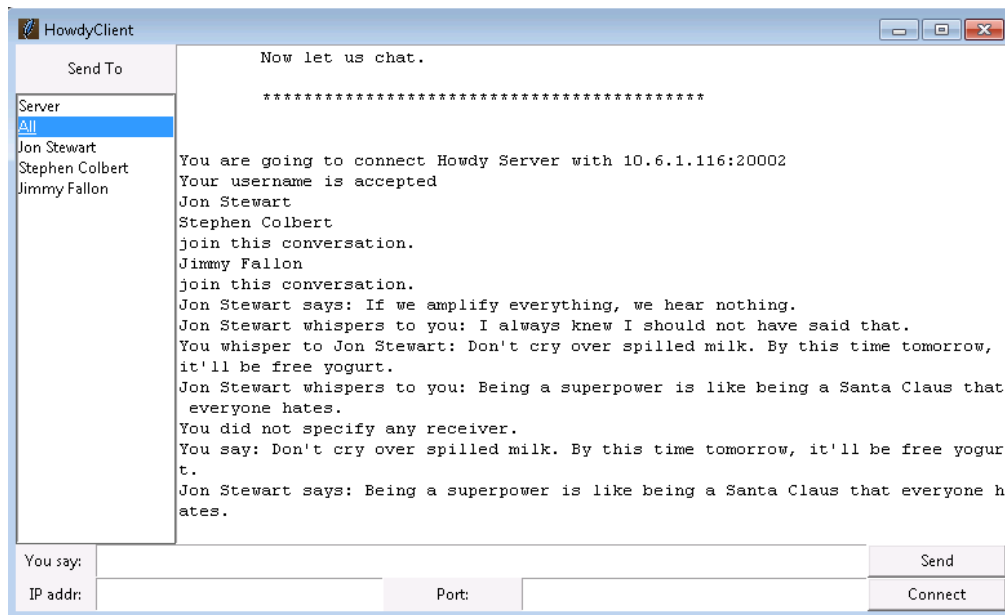
Texting While Receiving Message

Test Case Description			
This case is designed to verify the sending and receiving of messages on Howdy Client will not interrupt with each other.			
Pre-Conditions			
The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and all 3 clients are running and have connected to Win71 with unique username.			
Step	Actions	Expected Output	Success/Fail
1	On Win72, input 'Being a superpower is like being a Santa Claus' and select Stephen Colbert as receiver.		
2	On Win73, input 'Don't cry over spilled milk. By this time tomorrow, it'll be free yogurt.', select Jon Stewart as receiver and send	Message are received properly and words in text entry of Win72 is not interrupted.	Success
3	Quickly go back to Win72 and paste 'that everyone hates.' Then send the message.	Message can be sent to right receiver.	Success
4	Change receiver to 'All' and repeat step 1-3.	Message are received properly and words in text entry of Win72 is not interrupted.	Success
Test Result			
<p>Test result is success.</p> <p>No matter unicast message or broadcast message, the receiving of new message will not interrupt texting or sending of message.</p> <p>Test results are shown below.</p>			

Texting While Receiving Message on Win72



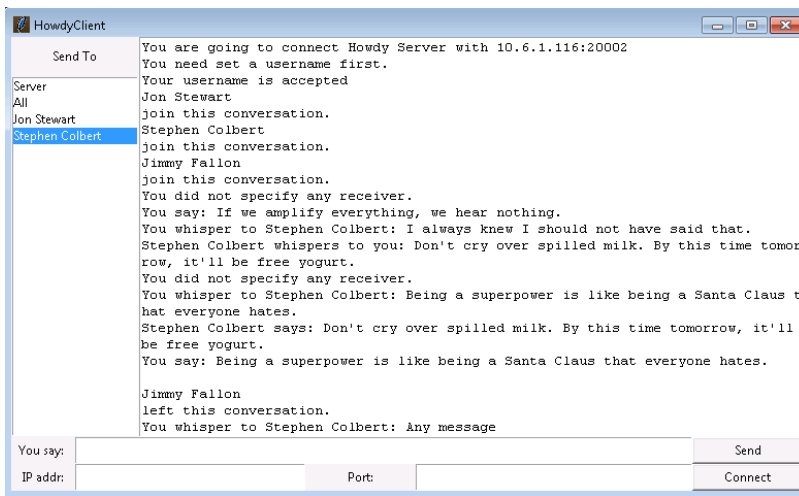
Texting While Receiving Message on Win73



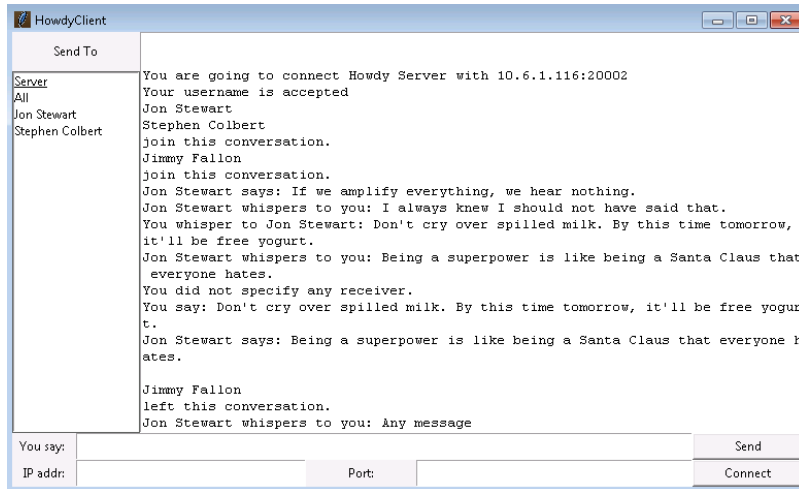
Single User Offline

Test Case Description			
This case is designed to show a user can logout without disruption of others.			
Pre-Conditions			
The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and all 3 clients are running and have connected to Win71 with unique username.			
Step	Actions	Expected Output	Success/Fail
1	Close Howdy Client on Win74.	On Win72 and Win73, there will be a message show that Jimmy Fallon left the conversation. On server, user logoff event will be printed out.	Success
2	On Win72, send any message to Win73.	The conversation will work as usual.	Success
Test Result			
<p>Test result is success.</p> <p>We can see after a user logout, server can quickly detect this event and print it out, also send out HCP message type 4 to all remaining users. All remaining clients will display relevant messages to users. The message exchange between rest users will not be disrupted.</p> <p>Test results are shown below.</p>			

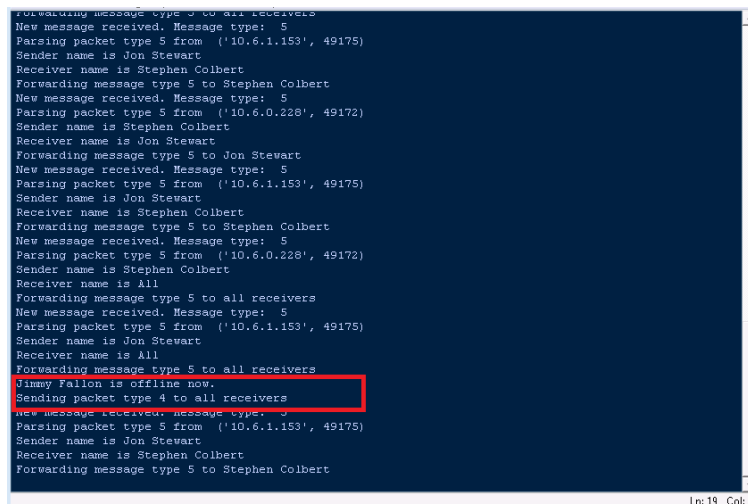
Single User Offline Win72



Single User Offline Win73



Single User Offline Event



Multiple User Offline

Test Case Description			
This case is designed to show multiple users even all users can logout without disruption of server or other clients.			
Pre-Conditions			
The server is running with IP address 10.6.1.116 and port number 20002 on host Win71 and all 3 clients are running and have connected to Win71 with unique username.			
Step	Actions	Expected Output	Success/Fail
1	Close Howdy Client on Win73.	On Win72, there will be a message show that Stephen Colbert left the conversation. On server, user logoff event will be printed out.	Success
2	On Win72, send any message to All.	The conversation will work as usual.	Success
3	Close Howdy Client on Win72.	On server, user logoff event will be printed out. Since there is no user online, server will print 'No user online now'.	Success
Test Result			
Test result is success.			

All User Logoff Events on Server:

```

Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Receiver name is Jon Stewart
Forwarding message type 5 to Jon Stewart
New message received. Message type: 5
Parsing packet type 5 from ('10.6.1.153', 49175)
Sender name is Jon Stewart
Receiver name is Stephen Colbert
Forwarding message type 5 to Stephen Colbert
New message received. Message type: 5
Parsing packet type 5 from ('10.6.0.228', 49172)
Sender name is Stephen Colbert
Receiver name is All
Forwarding message type 5 to all receivers
New message received. Message type: 5
Parsing packet type 5 from ('10.6.1.153', 49175)
Sender name is Jon Stewart
Receiver name is All
Forwarding message type 5 to all receivers
Jimmy Fallon is offline now.
Sending packet type 4 to all receivers
New message received. Message type: 5
Parsing packet type 5 from ('10.6.1.153', 49175)
Sender name is Jon Stewart
Receiver name is Stephen Colbert
Forwarding message type 5 to Stephen Colbert
Stephen Colbert is offline now.
Sending packet type 4 to all receivers
New message received. Message type: 5
Parsing packet type 5 from ('10.6.1.153', 49175)
Sender name is Jon Stewart
Receiver name is All
Forwarding message type 5 to all receivers
Jon Stewart is offline now.
No user online now.
Ln: 19 Col: 0

```

Source Code

HowdyServer.py



HowdyServer.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

'HowdyServer'
__author__ = 'Jingjun Zhang'

import socket
import threading
import re

BUFSIZE = 4096
MAXCONN = 10

class Chatconnection(threading.Thread):
    #create a dictionary mapping username and socket
    clients = {}

    def __init__(self, csock, caddr):
        #initiate thread of self and create a new object
        threading.Thread.__init__(self)
        self.cltsock=csock
        self.cltaddr=caddr

    def run(self):
        #start a thread which handle communication with a single user
        while True:
            # if user disconnects, server sould send update packet, remove coon from
            dict, and close conn.
            bmsg = self.cltsock.recv(BUFSIZE)
            # no more message received, remote client is closed.
            if len(bmsg) == 0:
                #define a temp variable to store name of offline user.
                offuser=''
                #finde the username of offline user.
                for u in Chatconnection.clients.keys():
                    if Chatconnection.clients[u]==self.cltsock:
                        offuser=u
                #offline user is in the list.
                if len(offuser)!=0:
                    #delete the connection from dictionary.
                    del Chatconnection.clients[offuser]
                    print(offuser+' is offline now.')
                    #send update packet to other users.
                    if len(Chatconnection.clients) != 0:
                        pkt = b'\x04'
                        for s in Chatconnection.clients.keys():
                            pkt = pkt + bytes(s, 'utf-8') + b'\x00' * (64 - len(s))
                        for c in Chatconnection.clients:
                            Chatconnection.clients[c].send(pkt)
                        print('Sending packet type 4 to all receivers')
                    #no other users online
                else:
                    print('No user online now.')
                #close connection
                self.cltsock.close()
            else:
                #close connection
                self.cltsock.close()
```

```

        if len(Chatconnection.clients) == 0:
            print('No user online now.')
        #stop running thread
        return

print('New message received. Message type: ', bmsg[0])
# receive message type 1, start username negotiation
if bmsg[0] == 1:
    print('Parsing packet type 1 from ', self.cltaddr)
    #extract username from message
    username = str(bmsg[1:65].rstrip(b'\x00'), 'utf-8')
    print('A username is received: ', username)
    # finde a new user, accept the new user, sending ack type 2 and update
type 4.

    if username not in Chatconnection.clients:
        #insert client socket into dictionary, and map it with user.
        Chatconnection.clients[username] = self.cltsock
        print('A username is accepted: ' + username)
        print('Accepted user comes from ', self.cltaddr)
        print('New user is added into list. Current User list: ',
Chatconnection.clients)
        #generate message type 2 and send.
        ack = b'\x02' + bmsg[1:65]
        self.cltsock.send(ack)
        print('Sending packet type 2 to ', self.cltaddr)
        #generate message type 4 and send.
        pkt = b'\x04'
        #insert user list into message
        for s in Chatconnection.clients.keys():
            pkt = pkt + bytes(s, 'utf-8') + b'\x00' * (64 - len(s))
        #send update user list to existing users.
        for c in Chatconnection.clients:
            Chatconnection.clients[c].send(pkt)
        print('Sending packet type 4 to all receivers')
        # username is in the list, reject the request, send a nak type 3.
    else:
        #generate nak message.
        nak = b'\x03' + bmsg[1:65]
        self.cltsock.send(nak)
        print('Sending packet type 3 to ', self.cltaddr)
# receive a regular message type 5, forward to proper destination.
elif bmsg[0] == 5:
    print('Parsing packet type 5 from ', self.cltaddr)
    #extract username of sender and receiver.
    sendername = str(bmsg[1:65].rstrip(b'\x00'), 'utf-8')
    recvername = str(bmsg[65:129].rstrip(b'\x00'), 'utf-8')
    print('Sender name is ' + sendername)
    print('Receiver name is ' + recvername)
    #sender is not registered, discard packet silently.
    if sendername not in Chatconnection.clients.keys():
        print('Error: Invalid sender name.')
        continue
    #sender is not in right connection, discard packet silently.
    if Chatconnection.clients[sendername] != self.cltsock:
        print('Error: mismatch of username and connection.')
        continue
    #message is send to all users.
    if recvername == 'All':
        #forward message to everyone except sender.
        for c in Chatconnection.clients:
            if Chatconnection.clients[c] != self.cltsock:
                Chatconnection.clients[c].send(bmsg)
        print('Forwarding message type 5 to all receivers')
    #message is send to a specific user.
    elif recvername in Chatconnection.clients.keys():
        #forward message to a specific user.
        Chatconnection.clients[recvername].send(bmsg)
        print('Forwarding message type 5 to ' + recvername)
    #error in username of receiver.

```



```

        else:
            print('Error: Invalid receiver name.')
            continue

    else:
        #server will not deal with any other message types, the packet should be
discarded silently.
        print('Error: Invalid message type.')
        continue

if __name__=='__main__':
    #default ip address and port number of Howdy server.
    srvip = '127.0.0.1'
    srvtcpport = 20002

    #ask user for IP address
    while True:
        strip = input('Please input a valid IP address of this Howdy Server, press Enter
to use 127.0.0.1.'+'\n')
        #if no input, use default value.
        if len(strip)==0:
            break
        #if user input is a valid ip address, update server ip addresss.
        if re.match('^\((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)\{3\}(25[0-5]|2[0-4][0-
9]|[01]?[0-9][0-9]?)$', strip)!=None:
            srvip=strip
            break
    #ask user for tcp port number
    while True:
        try:
            strport=input('Please input a valid TCP port number for Howdy Server, press
Enter to use 20002.'+'\n')
            #if no input, use default value.
            if len(strport)==0:
                break
            #update port number with user input
            srvtcpport = int(strport)
            #get a valid port num, break the loop
            if srvtcpport >= 1025 and srvtcpport <= 65535:
                break
            else:
                #get a invalid port num, restore default value.
                srvtcpport = 20002
        except:
            #if user input is not a number, keep asking for port number.
            continue

    srvaddr = (srvip, srvtcpport)
    print('You are going to start Howdy Server with ', srvaddr)

    #create main socket
    srvsock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #allow reuse address
    srvsock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    srvsock.bind(srvaddr)
    srvsock.listen(MAXCONN)
    print('Server is started and listening.')

    while True:
        #a client connection is established.
        sock, addr = srvsock.accept()
        print('A new client is connected: ', addr)
        #create a new thread for client connetion.
        newthread = Chatconnection(sock, addr)
        newthread.daemon=True
        newthread.start()

```

HowdyClient.py



HowdyClient.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

'HowdyClient'
__author__ = 'Jingjun Zhang'

import socket
import threading
import tkinter
import re
import sys

class MultiSockClient:

    def __init__(self):
        #buffer size
        self.BUFSIZE = 4096
        #default server address and port num
        self.srvaddr = '127.0.0.1'
        self.srvport = 20002
        self.mainsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        #all users/destinations for messages
        self.options=['Server', 'All']
        #local username
        self.myname=''
        #create the mainframe
        self.main_app = tkinter.Tk()
        self.main_app.title('HowdyClient')
        self.main_app.protocol('WM_DELETE_WINDOW', self.winclose)
        self.mainframe = tkinter.Frame(self.main_app)
        self.mainframe.pack()
        #add widgets into frame.
        self.labelop = tkinter.Label(self.mainframe, text='Send To')
        self.listboxul = tkinter.Listbox(self.mainframe, selectmode=tkinter.SINGLE)
        #add destination list into list box.
        self.optupdate()
        self.textboxmsg = tkinter.Text(self.mainframe)
        self.textboxmsg.insert(tkinter.END, '''
Howdy!!!!

Thank you for being a Howdy user.

Before sending any message, you should connect to Howdy server with valid IP
address and port number. If you do not specify any IP addresses and port numbers, you can
still connect the local server with port number 20002.

After connecting the server, the first thing you need to do is to send you user
name. Server and All are reserved so you cannot use them as user name. And if the name is
not accepted by the server, you have to choose another one till it is accepted. "Send To
Server" should be selected when you are sending your username.

While you are sending any message, you have to specify one option from options
list. You can send it to one user or to everyone.

Now let us chat.

*****

''')
        self.labelyou = tkinter.Label(self.mainframe, text='You say: ')
        self.entrymsg = tkinter.Entry(self.mainframe)
```

```

self.buttonsend = tkinter.Button(self.mainframe, text='Send')
self.labelip = tkinter.Label(self.mainframe, text='IP addr:')
self.entryip = tkinter.Entry(self.mainframe)
self.labelport = tkinter.Label(self.mainframe, text='Port:')
self.entryport = tkinter.Entry(self.mainframe)
self.buttonconn = tkinter.Button(self.mainframe, text='Connect')

#bind functions with two buttons
self.buttonconn.bind('<Button-1>', self.connectsrv)
self.buttonsend.bind('<Button-1>', self.msgsend)
#bind function to entry, so user can user enter key to send.
self.entrymsg.bind('<Return>', self.msgsend)

#create layouts
self.labelop.grid(row=0, column=0, rowspan=2, columnspan=8, sticky=tkinter.NSEW)
self.listboxul.grid(row=2, column=0, rowspan=18, columnspan=8,
sticky=tkinter.NSEW)
self.textboxmsg.grid(row=0, column=8, rowspan=20, columnspan=24,
sticky=tkinter.NSEW)
self.labelyou.grid(row=20, column=0, rowspan=2, columnspan=4,
sticky=tkinter.NSEW)
self.entrymsg.grid(row=20, column=4, rowspan=2, columnspan=24,
sticky=tkinter.NSEW)
self.buttonsend.grid(row=20, column=28, rowspan=2, columnspan=4,
sticky=tkinter.NSEW)
self.labelip.grid(row=22, column=0, rowspan=2, columnspan=4, sticky=tkinter.NSEW)
self.entryip.grid(row=22, column=4, rowspan=2, columnspan=10,
sticky=tkinter.NSEW)
self.labelport.grid(row=22, column=14, rowspan=2, columnspan=4,
sticky=tkinter.NSEW)
self.entryport.grid(row=22, column=18, rowspan=2, columnspan=10,
sticky=tkinter.NSEW)
self.buttonconn.grid(row=22, column=28, rowspan=2, columnspan=4,
sticky=tkinter.NSEW)
#start main GUI.
self.main_app.mainloop()

def winclose(self):
    #clear socket and GUI when user close window.
    try:
        self.mainsock.shutdown(socket.SHUT_RDWR)
        self.mainsock.close()
    except:
        print('No connection need to be terminated.')
    finally:
        self.main_app.destroy()
        sys.exit(0)

def connectsrv(self, event):
    # get ip addr and port num from user input
    iptemp = self.entryip.get()
    porttemp = self.entryport.get()

    try:
        # get valid ip addr and port number
        if re.match('^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.{3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$', iptemp) != None and int(porttemp) == self.srvport:
            # update ip addr and port num
            self.srvaddr = iptemp
            self.srvport = int(porttemp)
            self.textboxmsg.insert(tkinter.END, '\n' + 'You are going to connect
Howdy Server with ' + self.srvaddr + ':' + str(self.srvport))
            self.textboxmsg.see(tkinter.END)
            # connect with user specified ip addr and port num and start a new thread
            self.mainsock.connect((self.srvaddr, self.srvport))
            self.cltrecv = threading.Thread(target=self.msgrecv,
args=(self.mainsock,))
            self.cltrecv.daemon = True
            self.cltrecv.start()

```

```

        elif len(iptemp) == 0 and len(porttemp) == 0:
            # connect with default ip addr and port num and start a new thread
            self.textboxmsg.insert(tkinter.END, '\n' + 'You are going to connect
Howdy Server with ' + self.srvaddr + ':' + str(self.srvport))
            self.textboxmsg.see(tkinter.END)
            self.mainsock.connect((self.srvaddr, self.srvport))
            self.cltrecv = threading.Thread(target=self.msgrecv,
args=(self.mainsock,))
            self.cltrecv.daemon = True
            self.cltrecv.start()

        else:
            # fail to connect
            self.textboxmsg.insert(tkinter.END, '\n' + 'The IP address or port number
is not valid, please try again.')
            self.textboxmsg.see(tkinter.END)
            return

    except:
        #restore default value if exception happens.
        self.srvaddr = '127.0.0.1'
        self.srvport = 20002
        return

def msgsend(self, event):
    #get seq no of item in list box
    recvseq = self.listboxul.curselection()
    #check if user specify the destination.
    if len(recvseq)==0:
        self.textboxmsg.insert(tkinter.END, '\n' + 'You did not specify any
receiver.')
        self.textboxmsg.see(tkinter.END)
        return
    #get destination of message
    dest = self.listboxul.get(recvseq)
    #get outgoing msg from text entry.
    msg = self.entrymsg.get()
    #check if it is empty.
    if len(msg)==0:
        return
    print('You sent: ', msg)
    #convert message to bytes.
    bmsg = bytes(msg, 'utf-8')

    #send username to server
    if dest == 'Server':
        #stop sending anything if having a valid local username.
        if len(self.myname)!=0:
            return
        print('Sending username to server')
        #check length of username
        blen = len(bmsg)
        if blen>64:
            self.textboxmsg.insert(tkinter.END, '\n'+ 'The username is too long.')
            self.textboxmsg.see(tkinter.END)
        #send username to server
        pkt = b'\x01'+bmsg+b'\x00'*(64-blen)
        self.mainsock.send(pkt)
    #send msg to a single user
    elif dest != "All":
        #stop sending if do not have a username.
        if len(self.myname)==0:
            print('You need set a username first.')
            self.textboxmsg.insert(tkinter.END, '\n' + 'You need set a username
first.')
            self.textboxmsg.see(tkinter.END)
            return
        #display the message you are sending on client GUI.
        self.textboxmsg.insert(tkinter.END, '\n' + 'You whisper to '+dest+' : ' + msg)

```

```

        self.textboxmsg.see(tkinter.END)
        print('Sending message to ' + dest)
        #generate message to a single user.
        senderlen = len(self.myname)
        recverlen = len(dest)
        pkt = b'\x05' + bytes(self.myname, 'utf-8') + b'\x00'*(64-senderlen) +
bytes(dest, 'utf-8') + b'\x00'*(64-recverlen) + bmsg[0:self.BUFSIZE-129]
        self.mainsock.send(pkt)
        #send msg to all
    else:
        #stop sending if do not have a username.
        if len(self.myname) == 0:
            print('You need set a username first.')
            self.textboxmsg.insert(tkinter.END, '\n' + 'You need set a username
first.')

            self.textboxmsg.see(tkinter.END)
            return

        #display the message you are sending on client GUI.
        self.textboxmsg.insert(tkinter.END, '\n' + 'You say: ' + msg)
        self.textboxmsg.see(tkinter.END)
        print('Sending message to all')
        #generate message to all.
        senderlen = len(self.myname)
        pkt = b'\x05' + bytes(self.myname, 'utf-8') + b'\x00'*(64-senderlen) +
bytes('All', 'utf-8') + b'\x00'*(64-len('All')) + bmsg[0:self.BUFSIZE-129]
        self.mainsock.send(pkt)
        #reset text after sending a packet
        self.entrymsg.delete(0, 'end')

def msgrecv(self, conn):
    while True:
        # receive a msg
        try:
            bmsg = conn.recv(self.BUFSIZE)
        except:
            return
        #if length of message is 0, close socket.
        if len(bmsg) == 0:
            self.mainsock.close()
            return
        print('Raw message type: ', bmsg[0])
        print('Raw message: ', bmsg)
        #parse a username ack type 2.
        if bmsg[0] == 2:
            print('Parsing packet type 2')
            #update local username.
            self.myname=str(bmsg[1:65].rstrip(b'\x00'), 'utf-8')
            print('My username is', self.myname)
            self.textboxmsg.insert(tkinter.END, '\n'+ 'Your username is accepted')
            self.textboxmsg.see(tkinter.END)
            #parse a username nak type 3.
        elif bmsg[0] == 3:
            print('Parsing packet type 3')
            self.textboxmsg.insert(tkinter.END, '\n'+ 'The username is not accepted by
server, please try another one.')
            self.textboxmsg.see(tkinter.END)
            #parse a username update type 4.
        elif bmsg[0] == 4:
            print('Parsing packet type 4')
            #save original destination list.
            templist= self.options.copy()
            #number of names in the new list.
            numnames= (len(bmsg)-1)//64
            #clear list and add new users into the list.
            self.options.clear()
            self.options.append('Server')
            self.options.append('All')
            for i in range(numnames):
                s=str(bmsg[1+64*i:65+64*i].rstrip(b'\x00'), 'utf-8')

```

```

        self.options.append(s)
    print('Current options: ', self.options)
    #update window display.
    self.optupdate()
    #information of new user online and old user offline
    #find logoff users.
    offnum=0
    for u in templist:
        if u not in self.options:
            self.textboxmsg.insert(tkinter.END, '\n'+u)
            offnum=offnum+1
    if offnum>0:
        self.textboxmsg.insert(tkinter.END, '\n' + 'left this conversation.')
    #find logon users.
    onnum=0
    for u in self.options:
        if u not in templist:
            self.textboxmsg.insert(tkinter.END, '\n'+u)
            onnum=onnum+1
    if onnum>0:
        self.textboxmsg.insert(tkinter.END, '\n' + 'join this conversation.')
    self.textboxmsg.see(tkinter.END)
    #parse a regular msg type 5.
    elif bmsg[0] == 5:
        print('Parsing packet type 5')
        #get information of sender and receiver.
        sendername = str(bmsg[1:65].rstrip(b'\x00'), 'utf-8')
        recvername = str(bmsg[65:129].rstrip(b'\x00'), 'utf-8')
        #convert bytes to string message.
        msg = str(bmsg[129:].rstrip(b'\x00'), 'utf-8')
        #discard packets from invalid users.
        if sendername not in self.options:
            continue
        #handle boradcast message.
        if recvername == 'All':
            self.textboxmsg.insert(tkinter.END, '\n'+sendername+' says: '+msg)
            self.textboxmsg.see(tkinter.END)
        #handle unicast message.
        elif recvername == self.myname:
            self.textboxmsg.insert(tkinter.END, '\n'+sendername + ' whispers to
you: ' + msg)
            self.textboxmsg.see(tkinter.END)
        #handle invalide message.
        else:
            continue
    #clients discards other message types silently.
    else:
        print('Unknown packet type')
        continue

#update listbox
def optupdate(self):
    #get number of current items in the list.
    optsz = self.listboxul.size()
    #clear items.
    self.listboxul.delete(0, optsz-1)
    #update list with current destination list.
    for s in self.options:
        self.listboxul.insert(tkinter.END, s)
    self.listboxul.update()

if __name__=='__main__':
    startapp = MultiSockClient()

```