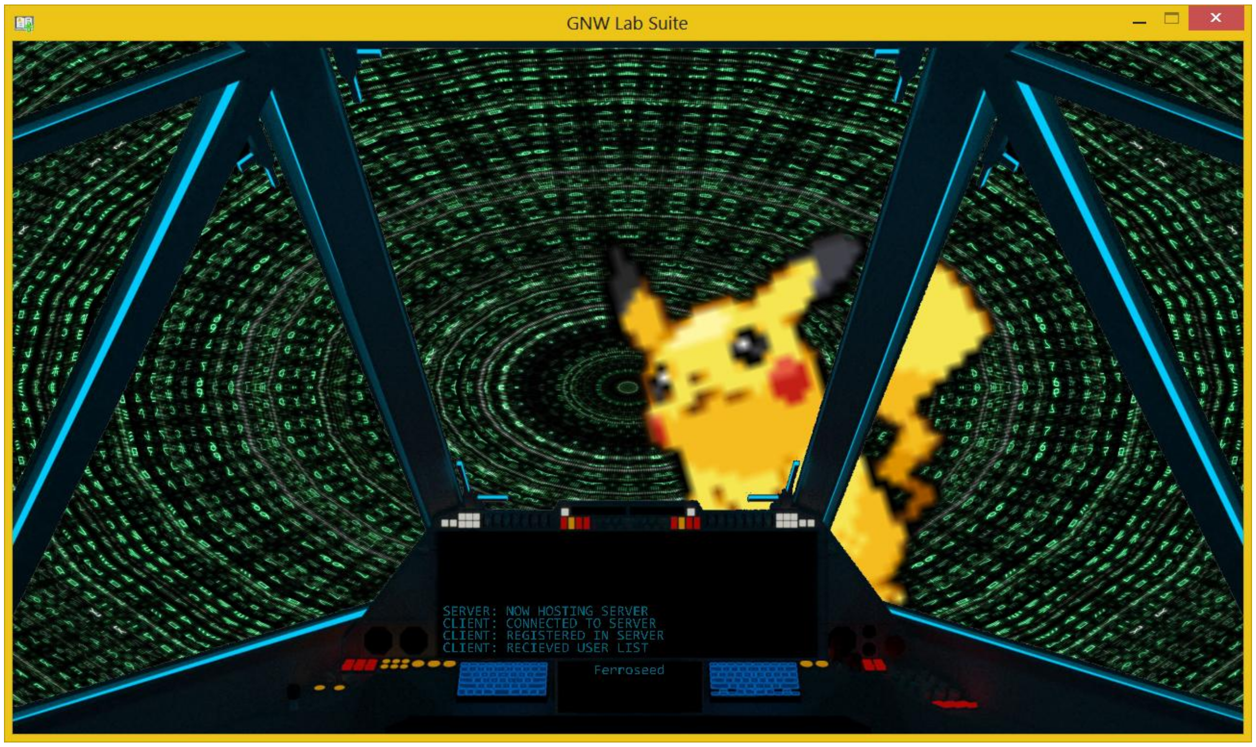
GNW Real Time Chat Lab



```
SERVER: NOW HOSTING SERVER
CLIENT: CONNECTED TO SERVER
CLIENT: REGISTERED IN SERVER
CLIENT: RECIEVED USER LIST
```

Ferroseed

Contents

It is encouraged that you read the Preface, About The Lab, Turn In & Potential Complications sections, before you start working on this software primarily described in the Instruction section. Use The Hyper Links to hop around in this document.

Preface

Writing a functional standalone network program isn't that hard, however things can get difficult when you need networking to run alongside and interface with other real time sensitive functionality such as a game or graphics application. Thought needs to be put into the design of such a program to be practical and efficient for both graphic and networking sake. This lab is a tutorial or guide for an implementation of introducing network related changes into an environment that is graphically running in real time.

Chat is a pretty standard feature known in online-internet gaming, and is an ideal candidate for the basics of what is described above. In this set of labs you will be given empty modules with nothing but an interface to make changes to the chat text that is rendered in real time alongside other things. You will be required to comply with the network message **Formats** described in this document so that your resulting application client or server will function alongside the example program provided. These message formats are listed at the bottom of this document. You will be required to do most of the API learning yourself, with what you learn during class lectures, using the lecture slides provided, and doing your own research using the msdn and internet.

The labs architecture provided is designed around using the TCP transport protocol and some of the challenges that come alongside it, such as separating network performance from real time game related performance. Injecting networking calls into a real time game can be detrimental to its performance in multiple ways. For one, networking API calls are usually blocking by default. This is designed around the asynchronous behavior of your computer network devices running alongside the majority of your program on the CPU. In summary, if you request a networking operation to happen, the active thread that the call is being made on will block until the conditions are fulfilled or fail for whatever reason. If you already had a single threaded game and you were to add blocking related networking between some of the logic, the program would only be as responsive as the networking performance. If the game requests data, it cannot continue until the data it requested arrives. This is why a simple fix for this problem can be to make the network calls not block, however this introduces a problem from the opposite angle. If a networking call was blocking, it is sure to deliver the request as soon as a low level interrupt resumes your application thread when delivering the results of the request to your application. When the network call blocking is turned off, if the network call request can't be completed, then the control is instead returned back to the application making the call. The application can then go about processing any other logic that needs to be done such as updates and real time rendering calls. The other angle issue here is that every time an incomplete net operation happens, it has to wait for other processing before it can attempt to make a request again. This is problematic for software that wants to show the results of networking as soon as it can. A real time networked program using non blocking TCP calls would also have to split up partial messaging on a frame to frame basis, because doing the full partial message loop described to you so far would almost be as detrimental as blocking, causing unstable frames and an inconsistent flow of the other real time operations.

The solution to these problems you might have guessed is to place the networking calls on separate threads and retain their ability to block. In fact blocking is good when dealing with other threads, because they are descheduled when the blocking conditions aren't met. This allows the CPU to focus performance to other threads and processes running. When any network condition is met, any operations that need to interface with the unhinged software running on the original thread can be immediately instigated. As well as no need to fragment partial messaging between other operations of the software. This is the architecture provided in this learning software.

The TCP transport protocol provides guaranteed in order delivery of the data. This is often important for certain types of data in games such as communication related operations like chat functionality. The UDP transport protocol doesn't hold these features over the data transport which can cause performance gain provided guaranteed in order delivery is not required. If UDP was being used and data needed to be guaranteed, your application layer would need to provide the understanding of this and potentially handle extra network operations to get the required data. Doing so can sometimes enact a performance degradation greater than the built in TCP protocol being performed at a lower level, especially during times of unstable network conditions, when networking that requires guarantees will need to play catch up. This lab is also designed to give you insight when attempting to choose the right transport related protocols depending on the networking problem. The chat functionality in this software is designed to give you an example of fitting efficient tcp operations to real time games. TCP is chosen because chat messages should not be dropped and should be delivered in order. When tcp operations are completed, the related data needs to be interfaced to the real time rendering. In this case it is done by thread safe writes to data being used by the real time operations.

About the Lab

The Main Menu

Secret Documentation!

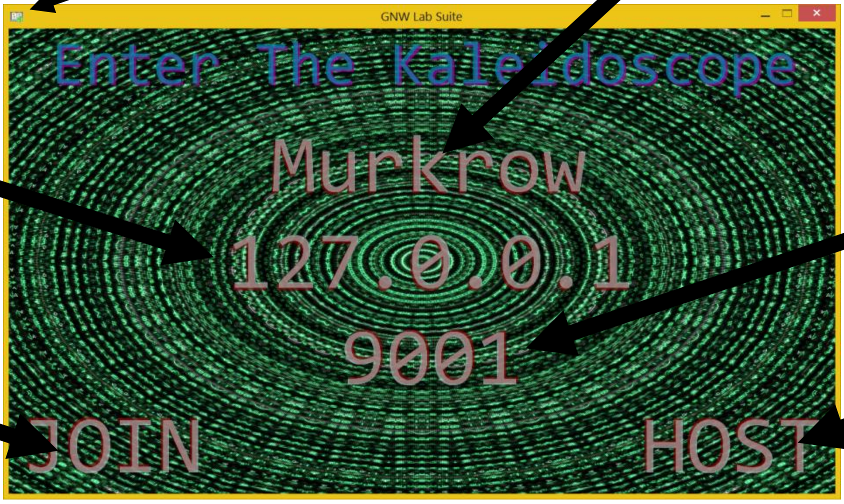
Set the user name.

Set the IP address that will be used by the client module when joining a server.

Set the port the server will be hosting at, and that the client will connect.

Starts a client module thread only.

Starts a server module thread, and a client module thread.



The Chat Lobby

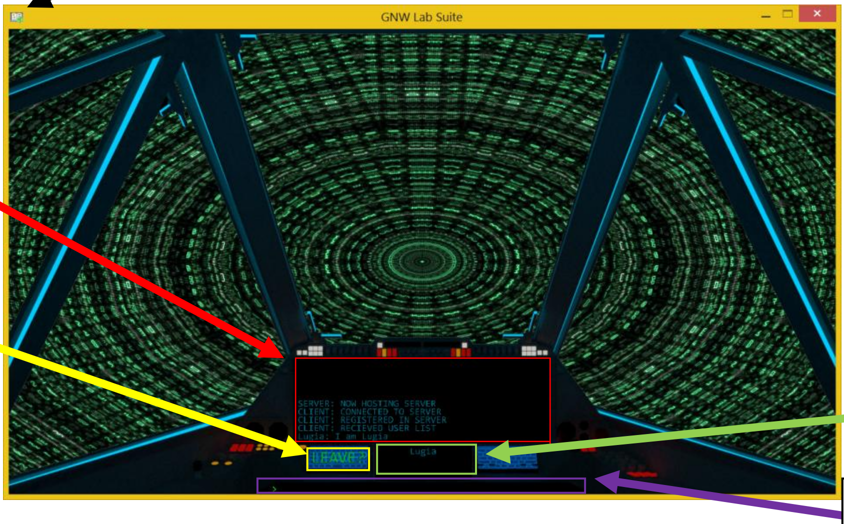
Secret Documentation!

Chat display area as added by the Chat Lobby Interface. Controls: (Left Click, Mouse Scroll)

Calls the stop method for any active network modules running. Returns to Main Menu. Controls: (Left Click)

Displays user list as added by the Chat Lobby Interface. Controls: (Left Click)

Displays unsent user typed chat messages. Controls: (Left Click, Enter, Up, Down)



```
SERVER: NOW HOSTING SERVER
CLIENT: CONNECTED TO SERVER
CLIENT: REGISTERED IN SERVER
CLIENT: RECIEVED USER LIST
Lugia: I am Lugia
```

The Display Interface Source Code

Attached to both the **TCPChatClient** and **TCPChatServer** is a reference to the **ChatLobby**. Listed Below are the 4 functions you will use as a result of your networking.

- **AddNameToUserList(string, id)** is used for adding a name to be rendered in the connected users box. Only the **client** should call this.
- **RemoveNameFromUserList(id)** is used for removing a name you have already added. Only the **client** should call this.
- **AddChatMessage(message, id)** is used for rendering a typed message associated with one of the users that has already been added to the user list. Only the **client** should call this.
- **DisplayString(string)** is used for displaying and signifying events such as a server disconnect, added/removed users and anything else that you might find helpful while debugging any problems. The **client** and the **server** should call this.

The **NetDefines.h/.cpp** is provided to define standard things that your networking modules can reference. Included is the enumeration of **NET_MESSAGE_TYPE** values that are used with the message Formats towards the end of this document. A **tcpclient** structure that can be used to Help describe clients in your implementation. Also a standard TCP partial message handler called **tcp_recv_whole()**. Be sure that you understand how this function works before using it and why it applies to the TCP streaming behavior and its delivery of data to your application.

Instruction

Now that the situation has been introduced from an analytical standpoint. Lab specific instruction can begin. You will be completing the client module of the software first. Use the example executable to host the chat server. Once your client has established a connection, you will then be sending and receiving specific message data with the server as outlined in this document. You will also be responsible for closing down the established communications at networking termination. Once finished, you will then move over to the server module and be implementing multiplexing logic designed around the management of multiple client I/O. Also reinforcement of the sending and receiving as you respond to the messages sent by your client and send back the messages received by your client.

Client instructions are as follows:

- Establish a connected TCP socket endpoint and send off the **register** message to the server in the client module **init** function.
- Begin the client **run** by receiving the generic message length header of 2 bytes.
- Continue by then receiving the remainder of the corresponding message as specified by the length.
- Parse the received message with the first byte which identifies which message type the received data represents.
- Send a close message to the server and then shutdown operation of and close the connection endpoint socket in the client **stop**.

Skip to the message Formats descriptions section to understand how to parse and react to messages sent by the server.**Error! Reference source not found.**

Server instructions are as follows:

- Begin listening at an address and port for a connection requests by creating a socket bound with address information in the server module **init**. Add this socket to a stored fd_set in preparation for multiplexing with the select function.
- Begin the server **run** by calling the select function with a copy of the stored set to see if any of the provided sockets are readable.
- If the listen socket originally added is ready for reading then accept a new socket endpoint representing a connected client. Add socket to stored set.
- If any connection end point socket is ready for reading then begin receiving the data on that socket the same as the client. Then parse the received messages and perform the corresponding server duties.
- Send a close message to the clients and then shutdown operation of and close the listen & connection endpoint sockets in the server **stop**.

Below is a description of the message formats.

Notice that every message has a 2 byte header "Length". This length header is always set to the value of the remaining corresponding bytes. For instance **cl_reg** below including the length can all be packed together which will total as one send of 20 bytes or any number of sends that add to 20 bytes, but the Length value is only in regards of the 18 remaining bytes that come afterwards. This is just a design choice. It would be a good idea to comply to this, because the example executables are designed to send and receive this way, and you can use them to gradually make progress. If you feel confident and you want to stray away from the standard outlined in this document, then you must communicate what you have in mind and have the ideas cleared with me. You will of course then be responsible for delivering source code that produces the same behavior as the example executables.

cl_reg

Length(uint_16) = 18	**Msg_type(uint_8) = cl_reg**	**User_Name(17 bytes array)**

The **cl_reg** message is sent to the server right after establishing a connection. The servers parse of the message will send back either **sv_cnt** representing a successful register or **sv_full** representing denial.

cl_get

Length(uint_16) = 1	**Msg_type(uint_8) = cl_get**

The client should request the list upon receiving the **sv_cnt** message from the server.

sv_cnt

Length(uint_16) = 2	**Msg_type(uint_8) = sv_cnt**	**Issued ID(uint_8)**

The server is only going to support 4 clients. If there is less than 4 clients connected when the server receives the **cl_reg** message from a new client. Then the server will add the issue the client into its system with an ID, and then send this message back to let the client know that they are registered.

sv_full

Length(uint_16) = 1	**Msg_type(uint_8) = sv_full**

If the server receives the **cl_reg** message from a client and already has 4 users considered to be registered. Then the server rejects the new client by sending this back to the client.

sv_list

Length(uint_16) = variable	**Msg_type(uint_8) = sv_list**	**Number in list(uint_8)**	**ID(uint_8)**	**Name(17 Bytes)**	**ID(uint_8)**	**Name(17 Bytes)**	**...**	**...**

The server sends this back the client upon receiving a request for it when getting a **cl_get** message from a client. The returned list should include all users considered to be registered by the server(including the same user that is requesting the list). It is up to you to manage the understanding of who is considered registered when implementing the server module. Upon receiving this message the client should populate the user list display with the ID name pairs using the **AddNameToUserList(string, id)** provided with the interface reference.

sv_add

Length(uint_16) = 19	**Msg_type(uint_8) = sv_add**	**User ID (uint_8)**	**User_Name(17 bytes array)**

After getting the **cl_reg** message from a client and accepting the client into the servers logic. The server will send all the already registered clients this message. Upon receiving this message those clients should add the ID name pair to the user list display using the **AddNameToUserList(string, id)** using the interface reference provided.

sv_remove

Length(uint_16) = 2	**Msg_type(uint_8) = sv_remove**	**User ID (uint_8)**

When the server receives the **sv_cl_close** message from a client, it needs to notify all other registered clients by sending this message. The server should also take care in removing this client from its management system new clients to register in the case that the server was full prior to this message being parsed.

sv_cl_msg

Length(uint_16) = variable	**Msg_type(uint_8) = sv_cl_msg**	**User ID (unint_8)**	**Null Terminated Chat Message(variable byte array) = Length - 2**

A client sends this message to the server which then sends it back to all connected clients including the one that sent it. Upon receiving this message all clients should then display this message to the chat display area using the **AddChatMessage(message, id)** provided in the chat lobby reference.

sv_cl_close

Length(uint_16) = 2	**Msg_type(uint_8) = sv_cl_close**	**User ID (uint_8)**

This message is sent if the stop functions are invoked on both the client or server net modules. If the server stop is called this message should be sent to all currently registered clients to notify them that the server is officially disconnecting. If the client stop is called it should send this message to the server. Upon getting this message should send the **sv_remove** to all other registered clients to let them know that this client has left.

Turn In

You should only be modifying the source files included in the **Chat Lab TODO** visual studio filter. Only zip and turn in source files that you have modified. Submit to the appropriate lab 2 & lab 3 turn in locations on Sidekick Moodle software.

Potential Complications

- Using namespace std in your project has a known issue to conflict with the windows sockets function bind. Please do not place the using statement, but instead std:: scope in any necessary data structures that you decide to include and use during the lab. This will avoid the problem.