JustChat

A Distributed Chat Room

CPSC 416 Final Project

Authors:

Haniel Martino, h1e9

Gabriela Hernandez, n5k8

Branden Siegle, w7z9a

Cindy Ngan, o6m8

Abstract JustChat is a distributed command line file sharing chat room where you can send public/private messages/files within the current group chat without the need to create an additional chat room to send private files or messages. In addition, JustChat allows new and existing clients to access publicly available files upon joining or rejoining.

1. Introduction/Motivation

JustChat is a distributed chat room system that provides global IRC communication in a simplified transparent way through the terminal/command line. JustChat enables participating clients, i.e. chat room members, to interact with each other in one seamless social setting, sending/receiving messages and files both publicly and privately.

The benefit of using JustChat is the simplicity it provides to the user in a chat room. JustChat is designed around three important features that a chat room should have: file sharing, privacy and sociableness. These three features are built on top of a distributed system utilizing RPC as the core design for the system. This approach provides a seamless interaction and minimizes the occurrence of system-wide failure.

A typical use case is among three friends Elon, Tony and Robert. The three guys sign up for a username and can instantly start interacting with each other through the command line, provided that the usernames they chose are unique and not taken by someone else. JustChat provides 6 simple commands for a user to follow so that they can privately message each other, and privately or publicly share files. Tony wants to tell Robert he has a surprise for Elon so he simply used the private message command to send the message only to Robert. Robert receives the message on his end, and Elon is none the wiser. Likewise, Elon wants to share a picture of his new car, a SELDOM 2016, to the entire chat feed, Elon uses the command for public file sharing and his picture is sent to Tony and Robert simultaneously. Tony and Robert is then alerted that Elon is sending a picture and JustChat asks them for permission to transfer the file which they both accept, in turn receiving a picture from Elon into their Download directory. Five minutes later, Natasha joins the feed and uses the command for available files. She sees that SELDOM_2016.png is available for download. She use the command to get the file, which is then downloaded into Natasha's Download directory.

As described above, JustChat enables a simplified yet novel approach to online chatting. This report will go into much more details about that interaction and how we made it possible. Section 2 will discuss the overall design and highlight roles among participating nodes within JustChat. Section 3 covers the implementation of those interacting nodes. It touches on libraries, methods used and structure of the code base. Section 4 addresses our evaluation of JustChat and our reasons for choosing a couple testing methodologies. We also discuss our use of ShiViz and how it visualizes the interaction among nodes within the system. Section 5 we discuss various limitations we faced on our system such as file transfer, command line errors and our reasons for using synchronous RPC vs asynchronous RPC. In Section 6 we reflect on our journey from start to finish, some difficulties we faced, and how we overcame them, such as redesigning our whole system from the ground up. Section 7 highlights the tasks each member in our group was responsible for, and finally, Section 8 gives a detailed overview of our RPC API and how the methods were used to interact within our system.

2. Design

a. Overview

The system is composed of three different types of nodes. A client node, a load balancer node, and a server node. The basic flow of the system is as follows:

A server node establishes a connection with a load balancer. A client node connects to a load balancer node in order to retrieve the RPC address of a connected server node. Next, the client node connects to the server node assigned by the load balancer and initializes the client-server relationship. The client is able to keep communicating with the server thereafter until failure, in which case the system follows appropriate recovery strategies. In case this connection sequence isn't met, recovery approaches are in place to allow the client to wait (block) until a server node is connected **(see Figure 1)**.

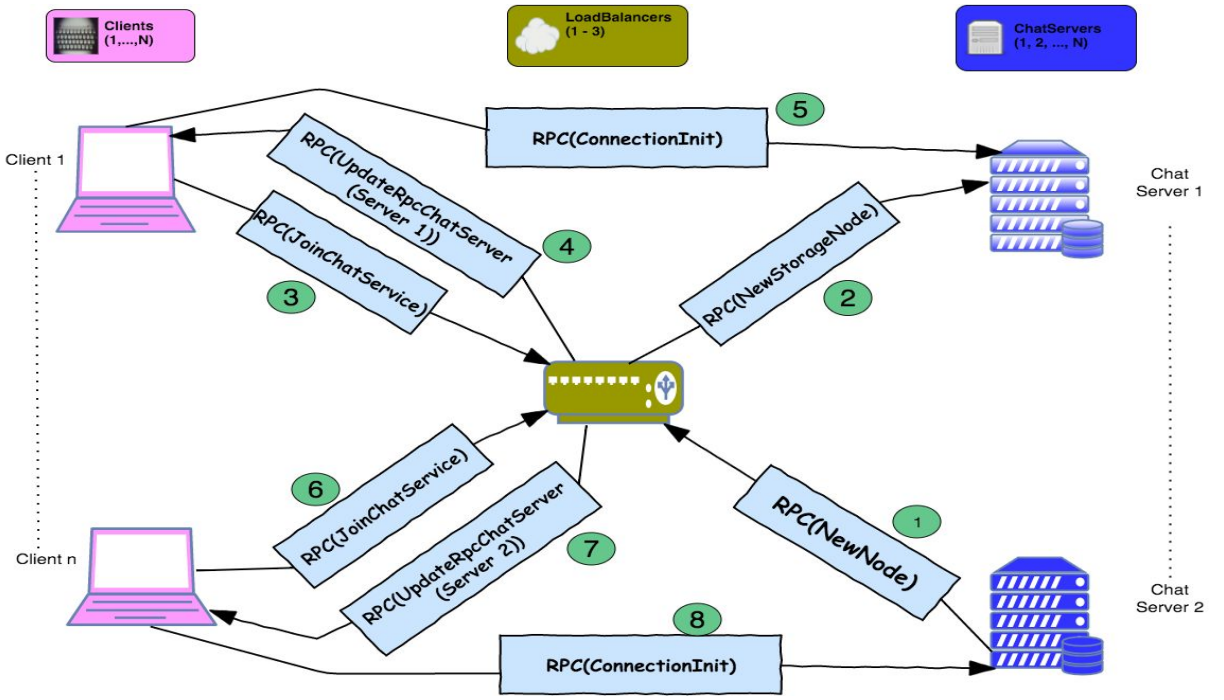

Figure 1 : Start up sequence for a trivial case (2 chat servers, n clients).
Description: Chat Server 1 and Load balancer is already online in this illustration.

b. Node Roles

In the system, all three types of nodes act as both a client and a server, depending on the current process. From here onwards, we will refer to the client nodes as clients, the server nodes as servers, and the load balancer nodes as load balancers.

c. Clients

The clients are an interface to the system. Their role is to provide a GUI to enable a user to use the system. The clients of the system are other hosts running the client.go file. Their identity is

known through their Username variable **(see Figure 2)** and RPC address that both the load balancer and server nodes keep the state of. This kept state is for communication between the client and these other nodes. The clients assume that there will always be a load balancer node and a server node available and that the API remains consistent throughout their time in the system. The clients rely heavily on RPC to interact privately with each other and to send public messages within the JustChat system. This approach allows for private messages to circumvent the servers and go directly to the intended recipient.

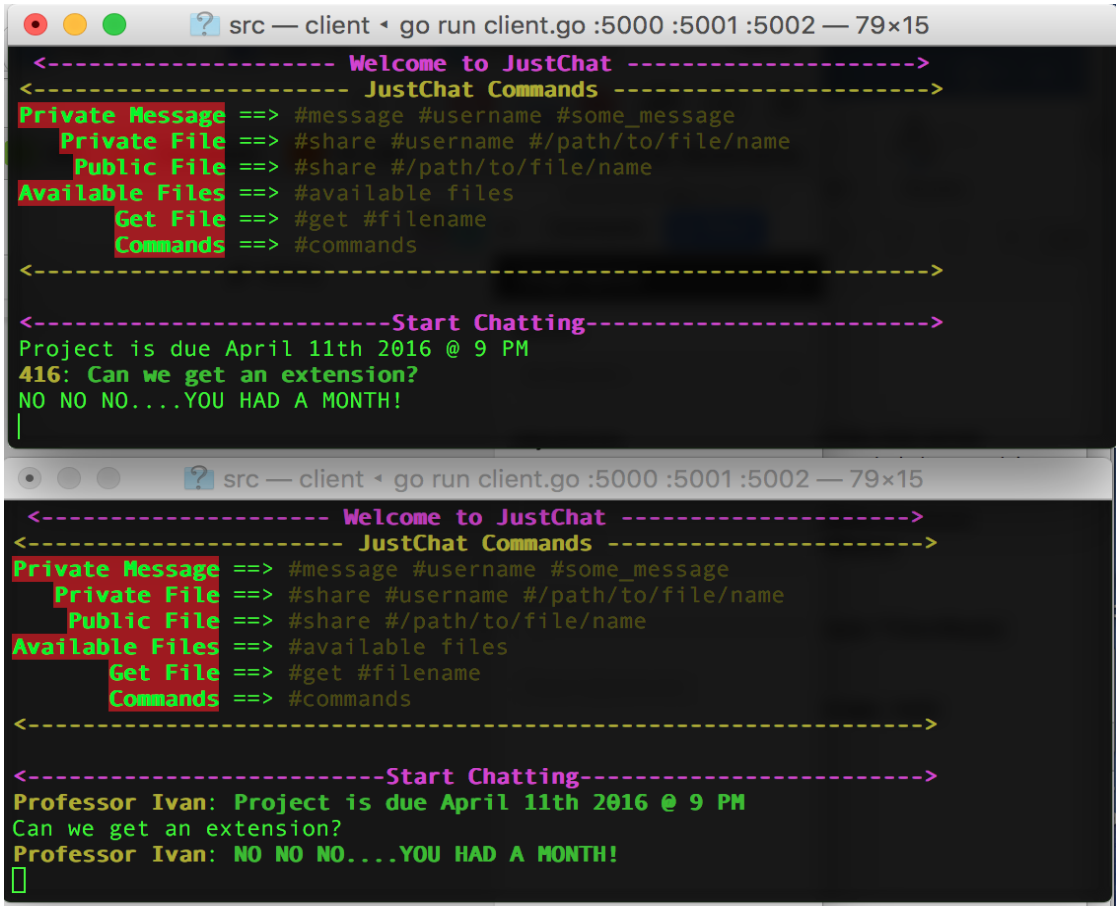

Figure 2: Client Interface
Description: A simple chat interaction between two users of the system, Prof. Ivan and 416.

d. Servers

The servers provide a pathway for messages and files to flow through in order to reach other clients in the system. Their purpose is to retrieve and send data to the appropriate nodes in the system, whether the nodes are other servers, their own clients or the load balancers. The servers of the system are hosts running the server.go file. Their identity is known through their RPC addresses and UDP Address that is kept both in the load balancers and other servers. The RPC addresses are used for client-server, server-server and load balancer-server communication while the UDP Address is used to ping other servers for their aliveness. Server

nodes assume that there will always be a load balancer to connect to and that the API will remain consistent throughout the process. The servers run concurrently and can block each other if they call a method that accesses the same data structure. See **Figure 3** for a client to chat room messaging example using the Server as an intermediary.

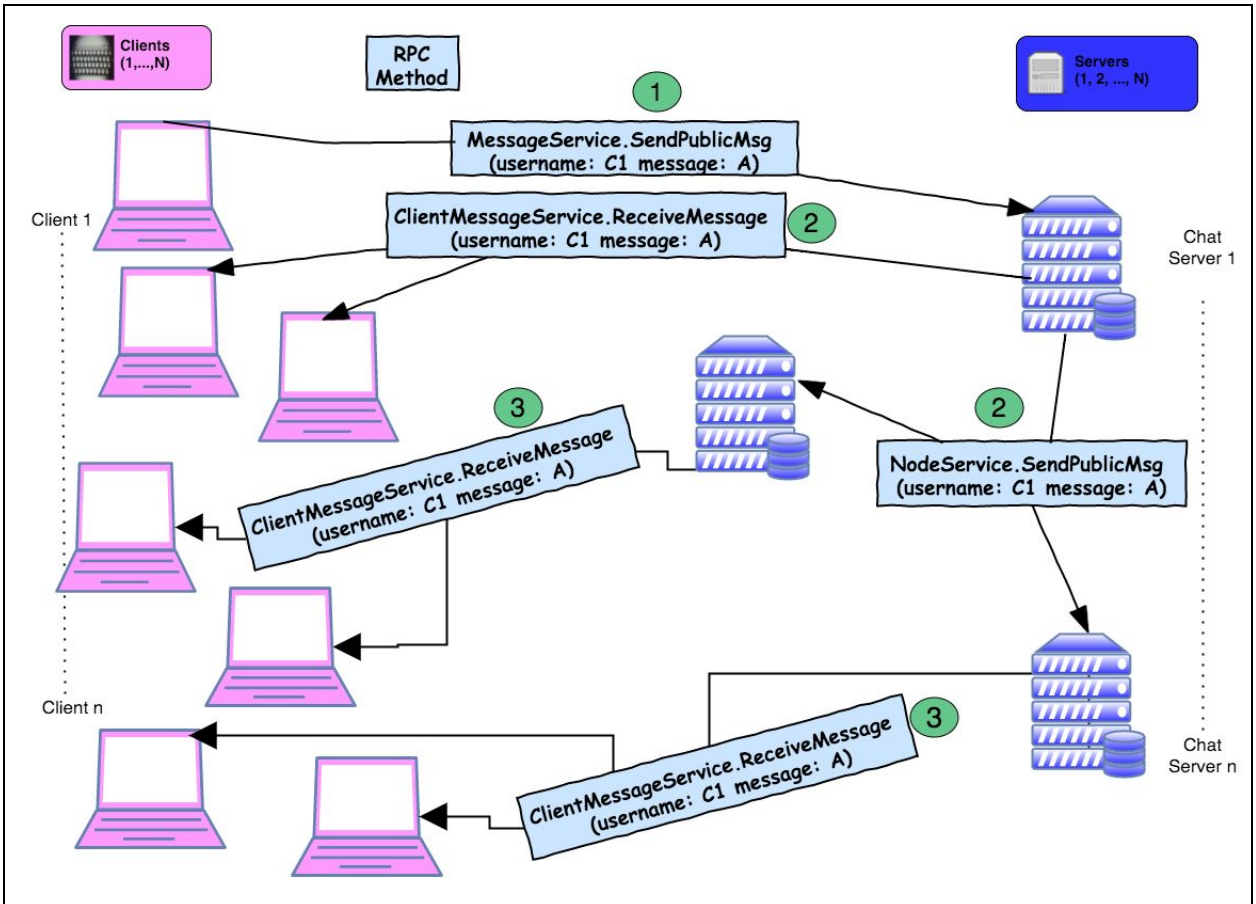

Figure 3: Client to Chat Room messaging
Description: A client sending a public message to the chat room. The same number indicates a concurrent process.

e. Load Balancers

The load balancers provide the clients with an entry point into the system. Their primary role is to authenticate users and present them with a server to connect to. Apart from this, load balancers must attempt to keep the system running on server failures. The load balancers are hosts running the loadbalancer.go file. Their identity is known to the user of the system through their RPC address. The servers and clients must know a load balancer's address in order to start up their respective files. Load balancers assume that storage nodes will eventually connect in order to provide clients with the system's intended behaviour and that the API calls will remain constant throughout the process. The load balancer is designed to handle restarts of any clients and message servers and provide them with the most up to date information at startup. Every load balancer maintains a list of all active servers, names of all available public files and a

list of taken usernames with their passwords. We are assuming that at least one load balancer will be running at any given point so this information isn't stored on disk.

The three roles are distinct and necessary due to our prescribed API of the system. The clients and servers need assurance that the load balancers will always be active once started since they are needed to start the client to server messaging. Therefore, the load balancer is intrinsic to starting the system. The servers are needed to handle the communication between clients, and the clients are needed to make use of the system. Although it is possible to merge some of these roles such as the load balancer and server, we decided against this so the clients only need to 'remember' three initial addresses of the three load balancers that distribute clients up to N servers. Figure 1 shows a simple approach to this, whereby client 1 joins the chat service, and the load balancer updates/gives the client a server's RPC address to initiate contact with.

f. Network

It is important to note that all the load balancers know about each other, in addition to all the servers, and all of the clients in the system. All of the servers know about each other, one load balancer each, and the subset of clients currently connected to it. A client knows about the load balancer and the server it is connected to. Even though some nodes do not know of each other in the system, there is some transitivity in how all nodes interact with each other (see Figures 2 and 3), which improves handling faults and expected behavior the system may face, for instance, failed servers or clients joining, leaving, and then re-joining the chat room.

- Load balancers:
    - Identified by the address that they are listening for RPC calls on
    - Store address information of clients and servers to update clients and for server information lookup by other servers and clients
    - Update address information of client when it rejoins and is authenticated
- Servers:
    - Identified their address information
    - Announce themselves to the load balancers when they initially connect to the system (see API)
    - Add client to its own client list and records address information
    - Differentiate clients by their username and address information
- Clients:
    - Identified by their unique username
    - All clients follow the required RPC protocols
    - All clients run the same implementation

g. System State

The complete, current state of the system is shared between the load balancers. They are aware of the current message servers, the list of client information, the status of all of the other load balancers, and a list of all available files. The other distributed state is the list of all active

message servers that is maintained by each message server; however they do not get a list of all clients or all files.

The clients do not maintain the state of the system aside from knowing the address of the current message server they are required to use. They do not know how many message servers there are or how many other clients there are.

h. Failures

Although JustChat recovers from a few failures, there are failures that are outside of the scope of what the system can deal with. For example, if there is a network failure and the load balancers become isolated then no new clients or message servers can join the system; however, any clients that were already connected can still communicate as long as their message server remains online as communication doesn't require a load balancer (the exception would be to retrieve a list of available files).

Nodes in the system itself can also fail. If there are only one active load balancer and one active server node, the system is greatly impacted if either of these nodes fail. However, if there are multiple load balancers and servers, a single failure doesn't have a huge impact. Load balancer failures have the greatest impact on the system since they are needed not only to start up the entire system but also to keep it alive when new nodes join. This is because they provide the clients and servers with necessary information. If all message servers do happen to fail, the clients and load balancers don't crash but simply block execution until a message server announces itself; at this point, all clients are assigned to this message server and communication continues. Through this, the system provides a graceful degradation and will be able to increase its performance once new message servers come online. Furthermore, there is no true fate sharing in that if one component fails, the whole system fails. The only fate sharing is that the system will block until there is, at least, one message server as described above.

i. Deviations From Original Proposal

There was some deviation in our final design from what was presented in our proposal. Most of the changes were regarding data storage in the server and their new dependence on the load balancers for user information.
The reason for this change was that it was more efficient to store the user data in the entity that would use it for authentication. Since the only entry point to the servers is through the load balancer, we knew that in order to get the most up to date information about users, the servers would have to query the load balancers, creating a new dependency on them. This dependency improves performance in cases for private messaging and private file sharing. Instead of a server querying multiple servers for a certain user, they can simply query one load balancer for this information.

We also gave the load balancers the duty of notifying other servers of new server nodes, since each server is aware of all other servers, the need to have a leader election algorithm is no longer existed.

Additional functionality such as client moderation and retrieval of chat history upon client startup introduced in our original proposal was not implemented due to lack of relevance to distributed systems and time constraints.

3. Implementation

a. Client.go

The client implementation in the JustChat system serves as the main entry point for users to interact with JustChat. Apart from the main method, client.go is sectioned off into four sections: RPC methods for the load balancer to call, RPC methods for the servers to call, RPC methods for clients to call, and helper methods that the main function and RPC methods utilize. The client also utilizes a clientutil package which does some of the heavy lifting such as editing output text, handling username and password input, and packaging files into byte arrays for transmission. One notable library used is the "golang.org/x/crypto/ssh/terminal". This library provides masking terminal/command line input from observers for when the user is prompted to enter their password.

In addition to the ‘ssh/terminal’ library, other notable libraries used were ‘net/rpc’, ‘sync’ and ‘bufio’ which provided core RPC interaction, locking/signaling for processes and command line input, respectively. The client made use of Go’s buffered channel feature to order incoming messages to the command line command line, that is, RPC methods that receive messages will simply send the message to the channel and then use a channel signal to indicate a new message is on the channel for output. This channel signal is used in a method flushToConsole() that runs concurrently with the main method.

b. Loadbalancer.go

The load balancer is the initial node that both the client and the server connect to. Upon startup, if there are already three other instances of load balancer running, the fourth load balancer will shut down. This is implemented using RPC calls to hard coded addresses of other load balancers. In the real world, this search for other load balancers would be done through a DNS lookup. The load balancer will then request the current state of the system and store that information in its global variables.

Once this setup is done, the load balancer will ping the servers every 20 milliseconds to ensure that they are alive. If a server is declared dead, then all clients that were connected to that server will be given a new server through an RPC call. When a client connects to the load balancer through the JoinChatService method, the load balancer will check to see if a server has been listed yet. If one has not been listed, execution will block until one is added; otherwise, the client will be given a server to connect to through an RPC call. When a new server connects to the load balancer, it is added to the list of possible servers to send clients to and is announced to the other load balancers. Finally, through the NewFile and GetFileList methods, the clients can learn which files have been made available by the servers.

d. Server.go

The server provides the clients with a passageway through which they can communicate. Given a load balancer's address and an IP:port string, the server will start up, start listening on randomly chosen ports, and then initialize a relationship with the load balancer. From the load balancer, the server retrieves a list of clients to serve as well as a list of the other servers on the system. The server pings all other servers through a UDP connection in order to be notified of server failures, in which case the server is deleted from the list. All other communication between the servers and the load balancers or clients is done through RPC. With clients, if an RPC connection cannot be initiated, the server assumes the client is offline and removes them from its global list of clients. In order for the servers to run as intended, it must be assumed that the load balancers will always be running. See API section for details on RPC.

4. Evaluation

To verify that JustChat is operating as expected, we used a few testing approaches and ShiViz to visualize and show message interaction.

a. Testing

In terms of system testing, we were not able to implement any unit/automated testing. This can possibly be ascribed to our overall lack of knowledge regarding the implementation of the Golang testing framework. However we were able to manually test our implementation with print statements to inspect correctness among the various interactions between clients, load balancers and servers. We were also able to view GoVector and chat history logs to ensure messages arrived in the correct order.

- **Commands** JustChat has a few commands that the client can use to interact with the system such as ones to send private messages/files, check available files, and send files publicly **(see Figure 4)**.

- **Correct RPC Addresses** Manually testing this using print statements was a good approach as we could verify that the correct RPC addresses were being sent to clients and, in turn, that those RPC addresses were correctly called and not altered on the client side.

- **Message Integrity** We also used print statements to verify message integrity after sending it to a server, and receiving it by a client. We verified that a message M being sent to a server S by client C was, in fact, the message M from client C.

- **Authentication** We tested authentication manually by first logging in with a correct username and password. We then logged out of the chat and logged back in again with the same username but a different password. Access was denied successfully. This is true even when the different clients attempt to log in to a different load balancer than previously connected to.

- **Message Ordering** This was tested similarly with message integrity, but we also verified this by clocking all messages to and from the server and writing to a chat history file, since all messages must pass through the server to the clients. This verification was done with up to three clients, one load balancer, and two servers, and thusly we must highlight that it may behave differently if a large amount of clients (100-1000) are connected.

Figure 4: Commands
Description: This shows the commands a user of JustChat have access to.

b. ShiViz

In terms of ShiViz integration, we logged the following events using GoVector:

- All RPC calls and their receipt in client, server, and load balancer.
    - If an RPC call replies with information that will be used, that interaction is logged as well, e.g. a client calling GetFile in server; we logged both the request and the subsequent file transfer, whereas for messages we only logged the single transfer and receipt.
- We did not log most RPC replies as the majority were used for error checking and thus were not a significant interaction in the system. We did not log pings to other nodes in the system as an aliveness check for the same reason.

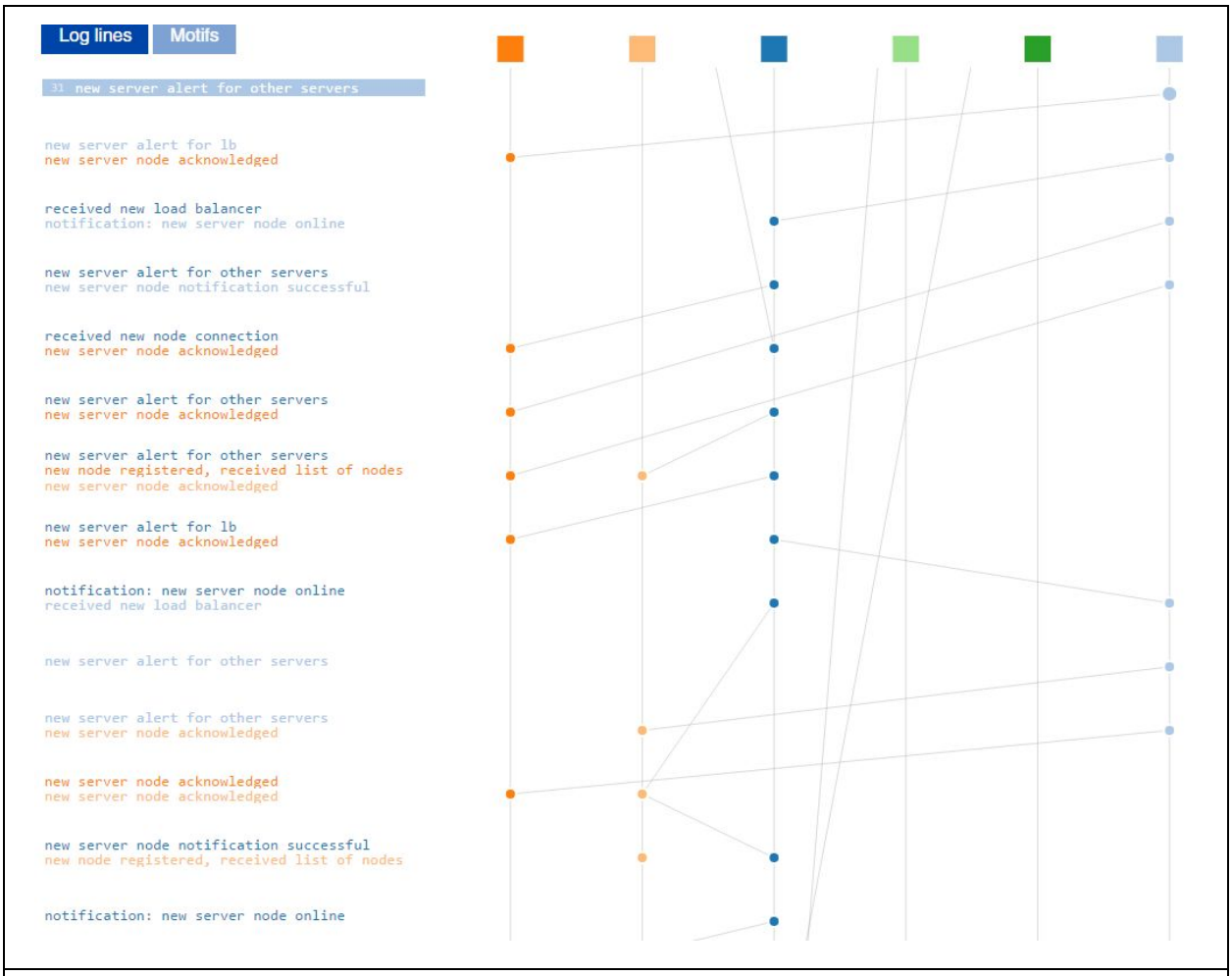

Figure 5: Part of a ShiViz generated graph of our system logs.
Description: Orange nodes are servers, blue nodes are load balancer, and green nodes are clients. Graph shows new load balancer and server joining the existing system.

5. Limitations

a. Synchronous RPC vs Asynchronous RPC

Our system utilizes synchronous RPC with most of our RPC procedures. This perhaps may have lead to a considerable slowdown in some procedures where the client (general client) is waiting on the server (general server) for it to return with a response. One notable aspect of this is in private messaging and private file transfer. The client makes a typical RPC to the server for the RPC address of another available client. The client waits until the server returns the RPC address of the other client. This process could have been implemented asynchronously whereby the client could proceed after making the call and continue chatting as per usual, and then be interrupted by the server when it finishes the RPC procedure.

b. File Transfer

File transfers to clients from other clients and servers are another notable feature that could be improved from its present design. Currently, a file transfer is initiated through the usual remote procedure call for both public and private files such that the client receiving the file has to approve receipt. Inherently this is great because clients can control with is written to their JustChat Download folder, but it fails with network bottleneck. Even before the client approves receipt of a file, the file is sent with the RPC call. A better design would have an initial RPC call asking for approval before sending the file to the client. If the client approves with this RPC call, then it would make another call to write the file to its Download folder. However, our present design was chosen because we assumed most clients would accept the file transfer.

c. Command Line (Terminal, Bash, Command Prompt etc.)

There is a reason chat rooms are best used in a GUI format where the user types a message into a text box, and it outputs into the chat window. Also, even while a user is typing a message in the text box, incoming messages are always being updated in the chat window. However, designing a chat room for a command line comes with a few limitations, one of which being the timing messages are output to standard output and the lack of a text box. We approached it in two ways but opted for the more real-time version. The first approach was to only show new messages after a user is finished writing to standard input and that user hits enter. However, this made the flow of the chat seem rather static and lacked that fluid-like nature seen in some chat rooms (i.e. FB messenger, iMessage, WhatsApp). Our second approach which gave users a more real-time feel was to continuously output messages to the command line upon receipt. However, we are using a command line, so the new messages will always be placed at the current cursor. This felt great if you were not typing a message, but was annoying when it inserted the new message at the current cursor point, cutting off your current message. The great thing was that you could still continue with your message and send it as is without losing its integrity, however for new users, there was a learning curve to understand the real-time-ness of JustChat.

6. Discussion

From the onset, we were excited to commit ourselves to an ambitious project with multiple features, but as we got further into the implementation and design phase, we realized that we should have limited our scope a bit and focus on core features of a chat room, like simply ‘just chatting’, but, that would be no different from the other 100+ chat services in existence. This motivated us to dig deep and ask ourselves, “What is missing from the various chat services we interact with daily?” Thus, JustChat was born.

One major challenge we ran into outside of implementation was team scheduling. Most of us lead pretty busy academic lives and this partially hindered our initial design phase. A subset of the group started working on the system independently with independent ideas of the design. This initial approach miraculously worked temporarily but we started to realize that it was a very

poor approach as one component was implemented with the assumption that the other component will play by its rules.

Fast forward to a couple weeks before the project is due and a team meeting with a TA, resulting in another meeting with the professor. After these meetings, we were quite certain that approaching JustChat through an RPC API was the best approach. Admittedly, we may have used this RPC hammer a bit too much, but it has served its purpose well. Using RPC allowed us to agree on how the nodes in the system will communicate with each other, irrespective of their own internal implementation.

The transparency RPC provided helped us form contracts between clients, load balancers and servers. This transparency allowed the interacting nodes to be aware of the others architecture (416 Lecture, January 13th) and the contracts served as the main gateway for communication among all these participating nodes in JustChat. It also allowed us to handle partial failures of our system (see Failures) by utilizing an “at-least-once” approach whereby we kept retrying on the client side until we got a response. We utilized this for clients sending public messages and making requests for another client's information. Some of our other approaches took on a strawman approach where we had the client block until the server was responsive. This design should have considered a time-out for periods that may exceed a certain time frame. We used this for clients on initial login if all servers are offline.

Another another issue we faced stemmed from testing on different platforms. For instance, we realized that Windows and Macs operating systems treat newline characters differently. Windows has a carriage return (‘\r’) whereas Macs do not. This caused some very unusual output on the command line from Macs to Windows. Messages from the Mac showed ‘perfectly’ on the command line, but when messages were received from a Windows PC, it was garbled. Implementing JustChat made us realize the importance of handling newline characters across platforms. This particular problem set us back by at least 2.5 hours trying to debug the problem.

Concurrency also caused a major issue when an external entity called an RPC method on your node while you were interacting with the command line. This issue was most prevalent in the client whereby an RPC method runs in a separate thread than the main thread. Two of the RPC methods allows for command line interaction where the user is asked permission to transfer a file. This RPC method can be called anytime by a client/server. When it is called, it is waiting for a response from the user. However, the command line is also waiting for an entry for a chat message to send as a public message. When the user thinks they are responding to the RPC prompt, such as “Would you like to download this file? (Y/N)” and the user enters “Y”, that Y is sent as a public message instead of a response to the RPC prompt. The user then would have to enter “Y” again when the RPC method has ‘control’ of the command line. This caused a back and forth behavior between the main thread and the RPC method, where the first entry is sent to the public chat, and the second entry is received by the RPC method. We solved this by adding an input lock to the command line, where the RPC method waits until the main method is finished using the command line and grabs the lock to process the user’s response.

7. Allocation of Work

- *Haniel Martino* - Implemented the client code (including clientUtil) and wrote the API for client.
- *Gabriela Hernandez* - Implemented the server code and wrote the API for server.
- *Branden Siegle* - Implemented the load balancer code and wrote the API for load balancer. In addition, aided in the usage of locks for go routine synchronization on the client, load balancer and server.
- *Cindy Ngan* - Integrated GoVector on client, server and load balancer, generated logs using ShiViz. Ensured message ordering on the server.

8. APIs

a. client.go

Service: ClientMessageService

UpdateRpcChatServer(args *ChatServer, reply *ServerReply) error

This method is called by a load balancer to update the client with a new server to send messages and make requests to. This method is called after the client first connects to just chat and is authenticated. If the server the client is sending messages to fails, the load balancer updates the client with a new server.

The ChatServer type contains values for the address information that the client will use to make RPC calls to the message server, and the messaging servers name.	The ServerReply contains a string message which is usually left blank indicating success.

TransferFile(args *FileData, reply *ServerReply) error

This method is called by a server when a public file is sent to the chat feed. The server sends the file to the clients and then ask for permission to store it, if client approves, the file is stored in the client's Download directory and replies with Received. Otherwise, client rejects and replies with "Declined".

The FileData type contains the username of the file owner, the name of the file, the size of the file and the byte array of the data contained in the file.	The ServerReply from the client contains either "Received" upon file receipt or "Declined" upon rejection or any file handling errors.

ReceiveMessage(args *ClientMessage, reply *ServerReply)

This method is called by a server when a public message is available for the client to see from the chat feed.

The ClientMessage contains the username of the message owner and the message itself.	The ServerReply contains a string message which is usually left blank indicating success.

TransferFilePrivate(args *FileData, reply *ServerReply) error

This method is called by another client when they want to send a private file. The client sends the file to the clients and then ask for permission to store it, if client approves, the file is stored in the client's Download directory and replies with Received. Otherwise, client rejects and replies with "Declined". Replies are then shown to the sending client on their console.

The FileData type contains the username of the file owner, the name of the file, the size of the file and the byte array of the data contained in the file.	The ServerReply from the client contains either "Received" upon file receipt or "Declined" upon rejection or any file handling errors.

ReceivePrivateMessage(args *ClientMessage, reply *ServerReply) error

This method is called by a client when sending a private message to another client.

The ClientMessage contains the username of the message owner and the message itself.	The ServerReply contains a string message which is usually left blank indicating success.

b. loadbalancer.go

Service: NodeService

NewFile(*string, *string) error

This method will add the file name to the list of globally available files and return "SUCCESS" upon completion.

The first string is the name of the file that is available to request.	The second string is a response message indicating success.

NewNode(*NewNodeSetup, *NodeListReply) error

This is the first call a new server will make when it comes online. This method will add the server to the list of available servers and will signal to any waiting routines that there is a new server added. All running load balancers are alerted to the additional server.

The NewNodeSetup type contains values for the address information that clients will use to make RPC calls to the server, the address information that the load balancers will use to make RPC calls to the server, and the address that the message service will be listening for heartbeats on.	The NodeListReply can either contain a pointer to a ServerItem or a pointer to a linked list of ServerItems

GetClientAddr(*ClientRequest, *ServerReply) error

Given the username of the client in a ClientRequest message, the load balancer will return the public address that RPC calls can be received on for that client.

The ClientRequest contains a string for the username of the client for which the address is being requested.	The ServerReply contains a string which is a message sent back to the client.

Service: LBService

NewNode(*NewNodeSetup, *NodeListReply) error

This method is called by another load balancer to alert this load balancer of the addition of a server to the system.

The NewNodeSetup type contains values for the address information that clients will use to make RPC calls to the server, the address information that the load balancers will use to make RPC calls to the server, and the address that the message service will be listening for heartbeats on.	The NodeListReply can either contain a pointer to a ServerItem or a pointer to a linked list of ServerItems

UpdateClient(*NewClientObj, *NodeListReply) error

This method updates the clientList to reflect changes when a client reconnects to a different load balancer with the same username and password. The listed client's address for RPC calls will be updated to the latest value in the NewClientObj.

The NewClientObj can either contain a pointer to a ClientItem or a pointer to a linked list of ClientItems	The NodeListReply can either contain a pointer to a ServerItem or a pointer to a linked list of ServerItems

NewClient(*NewClientObj, *NodeListReply) error

This method is called by another load balancer to alert this load balancer of the addition of a client to the system.

The NewClientObj can either contain a pointer to a ClientItem or a pointer to a linked list of ClientItems	The NodeListReply can either contain a pointer to a ServerItem or a pointer to a linked list of ServerItems

GetCurrentData(*LBMessage, *LBDataReply) error

This method alerts this load balancer that another load balancer is online and replies back with the current list of clients in the system and the current list of servers in the system. The status of the load balancer which sent this message will be updated to now be 'online.'

The LBMessage contains a string which is a message that indicates if data needs to be sent back or not and an integer value of the load balancer which is sending the message	The LBDataReply contains the current state of a load balancer. The values are pointers to a linked list of ServerItems and ClientItems

Service: MessageService

JoinChatService(*NewClientSetup, *ServerReply) error

This is the first method a client calls when it comes online. It will add the client to the list of clients after authenticating username and password. Then it will find a chat server to use and tell the client to connect to it.
***Calls are then made to other load balancers to alert them to the changes.*

The NewClientSetup contains values for the username, password, and address for RPC calls of the client making the call.	The ServerReply contains a string which is a message sent back to the client.

GetFileList(*string, *([]string)) error

This method returns a list of all the filenames currently available for download from a server.

The first string is a generic message that can be any value and is unchecked.	The string array is a list of all available filenames that can be requested.

c. server.go

Service: NodeService

NewStorageNode(*NewNodeSetup, *ServerReply) error

This method is called by a load balancer to alert this server of the addition of a message server to the system.

The NewNodeSetup type contains values for the address information that clients will use to make RPC calls to the message server, the address information that the load balancers will use to make RPC calls to the message server, and the address that the message service will be listening for heartbeats on.	The ServerReply contains a string that is used to indicate success.

SendPublicMsg(*ClockedClientMsg, *ServerReply) error

*This method is called by another server and sends the message of struct type *ClientMsg found within the *ClockClientMsg to all of the clients this server is connected to. If a connection to a certain client is not able to be made, the client is deleted from the global list of clients connected to this server.*

The ClockedClientMsg contains a ClientMsg that this server will forward to all the clients that are connected to it, a ServerId string that is this server's UDP Address, and a Clock int used for timestamps.	The ServerReply contains a string that is used to indicate success.

SendPublicFile(*FileData, *ServerReply) error

*This method is called by another server and sends the file of struct type *FileData to all of the clients that this server is connected to. If a connection to a certain client is not able to be made, the client is deleted from the global list of clients connected to this server.*

The FileData contains a Username string which the server uses as a message field, file statistics in the form of FileName and FileSize strings, and the file data itself.	The ServerReply contains a string that is used to indicate success.

StoreFile(*FileData, *ServerReply) error

*This method is called by another server and stores the file of struct type *FileData in the directory ./Files/ of this server.*

The FileData contains a Username string which the server uses as a message field, file statistics in the form of FileName and FileSize strings, and the file data itself.	The ServerReply contains a string that is used to indicate success.

GetFile(*String, *FileData) error

This method is called by another server and checks to see if it has the file with the specified name. If it does, it returns the file and if it doesn't, it returns with "404" in the Username field.

The first string is the name of the file that is expected to be retrieved.	The FileData contains a Username string which the server uses as a message field, file statistics in the form of FileName and FileSize strings, and the file data itself.