# Project Proposal:
# JustChat

Team Name: 416 Squad Goals

Team Members:
Haniel Martino, h1e9
Gabriela Hernandez, n5k8
Branden Siegle, w7z9a
Cindy Ngan, o6m8

February 29, 2016

## Table of Contents

## 1. Problem

Online text-based chat rooms have been used as a means of convenient communication for the latter part of the 20th century and have been revolutionized in recent years (Slack, FB Messenger, Skype, Google Hangouts etc.). However, many of these chat rooms lack one convenient feature: sending private messages/files within a current group chat feed, without the need to create an additional 'room'/channel/chat to send that file or message. In addition, allowing new clients to access publicly available files upon joining.

## 2. Overall System Description

JustChat is a distributed chat room system that aims to provide 'global' IRC communication in a simplified way through the terminal/command line. Our main goal with JustChat is to enable 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.

## 3. Proposed Solution

### 3.1 Client Logic

JustChat intends to provide a file transfer interface where participating clients can send and receive files through the chat room (more on this to follow) to/from other participating clients without other participating clients being aware of this file transfer. JustChat also serves as a general chat room where multiple clients can interact with each other in a group like manner. Clients can join and leave from the chat room continuously. Each client's message will have their username assigned to it to differentiate between participating clients. Clients in the chatroom will be aware of all participating clients by the client's username.

### 3.2 Client Moderation

Among participating clients, one moderator will be selected by a 'first-to-join' protocol, that is, if a client is the first to join the chatroom, they automatically become the moderator of that chat room. There can only be one moderator of any chat room and this moderator role is assigned by the main routing server that keeps a chronological list of all the connected clients. If a moderator fails, the least recently joined client becomes the moderator of that chat room. A moderator is given a timeout before it's considered failed, and its role will be reassigned to the next client in line. The server sends a message to the new moderator, assigning it the role as moderator. The moderator role comes with one simple task: the ability to kick other participating clients out of the chat room as they see fit.

To access the chat room, each participating client must have knowledge of the chat room's password/key and location(ip:port). Each client has to choose a unique username upon joining.

## 3.3 Client Failure

Each client continuously sends a heartbeat to the main routing server. This server assigns a timeout to each client to receive the client's heartbeat. If a client does not send a heartbeat within the timeout window, the client is deemed to have failed. The routing server then removes the client node from the connected client's buffer, update the chat room that "failed-client-username has left the chat room", and then closes any connections from that client. Failed clients can reconnect but they lose all moderation status, unless they are the first-to-join again.

## 3.4 Client's Behavior After Server Failure

The main routing server will have a static address, hence, upon server failure, the clients will continuously try to reconnect to the server. The state of previous connections is saved on one of the replicated servers therefore a moderator will retain their role upon failure, and the connected queue remains the same. All timeouts will be reset upon server failure so clients will have the chance to retain their position in the queue within the set timeout window.

## 3.5 Decentralization of Server Storage

The chat server's storage will be distributed among three different types of server nodes. Each server node will specialize in storing one of 3 types of useful data for the system. Specifically, they will either store user data, chat history, or files that have been transferred. The server will have k nodes of each specialized type of store, where data will be copied x amount of times, in order to reduce the server's probability of data access failure. Apart from this, and in case of total server failure, the server nodes will write their data to disk in order for the data to persist.

## 3.6 Different Types of Data Stores

The three different types of data stores are user data, chat history, and files transferred. Although none of the following is set in stone, we have decided to keep the following useful data:

- User data will consist of keeping things like a username, IP:PORT, time of last disconnect, a list of files transferred (upload/download lists), etc. It is important to note that usernames will be assumed distinct, in order to map users to their chat history.
- Chat history will store a chat ID, timestamps, usernames of users involved, and the actual text.
- Files transferred will keep information like the uploading user's information (username, IP:PORT, type of connection, etc.), as well as the type of the file, and a timestamp displaying when it was first uploaded for transfer.

## 3.7 Server Nodes Leader Election Algorithm

Each type of server node will have its own leader election algorithm in order to further modularize our system. That is, each type of store group (user data stores, chat history stores, and files transferred stores) will run its own leader election algorithm. Therefore, there will always be 3 distinct leaders out of all of the nodes.

The way the leader will be chosen is based on whichever store joined the server first. Incase of leader failure, the next node (of the same type) to join will become leader, and so on.

## 3.8 Handling Server Front-End Failure

Since the server stores its data in a distributed fashion, each server node will have to handle the possibility of front-end failure. To do this, each server node will continuously attempt to reconnect with the front-end as soon as its ongoing connection is terminated. This will be done to minimize data downtime and to get the server running as quickly as possible.

### 3.9 Server Logic

When a user initially starts up a messaging client, the server will add the connection to a list of client objects. A client object is defined below:

struct Client{ *net.conn: connection, string: username}

Next, the messaging client will ask the user to provide a username. This must be unique and the server will make sure it is; if the username has already been taken then the server will send the appropriate response and the user will have to select a different username. The client will get a message that will indicate the username has been assigned to it. This will be done atomically to eliminate the possibility of two clients having the same username. When a client disconnects, this is announced to all clients in the chat and the username is free again after a set period of time. This delay will make sure that no private messages which are delayed are sent to a different user who scoops up the old username quickly.

When a user types a message and sends it out, the server will determine if it is one of three types of messages: public message for all clients, private message for specific client, or command for some operation. If it is a public message, the server will broadcast the message out to all of the other connected clients and save the message in the chat history. If the message is private for another specific user, the message will only be sent to the user specified by the username. If there is nobody with the username specified an error will be sent back. Lastly, the command message will be reserved for things like cleanly disconnecting, sending files, or for the moderator to kick out a specific user. See below for commands.

As all messages will have vector timestamps, we can use these for sorting the chats as they come to the clients. These time stamps will also be used to generate our output for ShiViz.

Our server will replicate the chat logs, publically available files to be retrieved, and current client list to several different backups. This way if the server goes down, the state will not be lost and when the server is back online, the clients can then re-attempt to connect to it. Also, we want the ability for a newly joined client to be able to see the public chat history. If the server restarts and then a new client joins, we want that new client to have the chats sent to their terminal.

<u>3.10 Chat Commands</u>

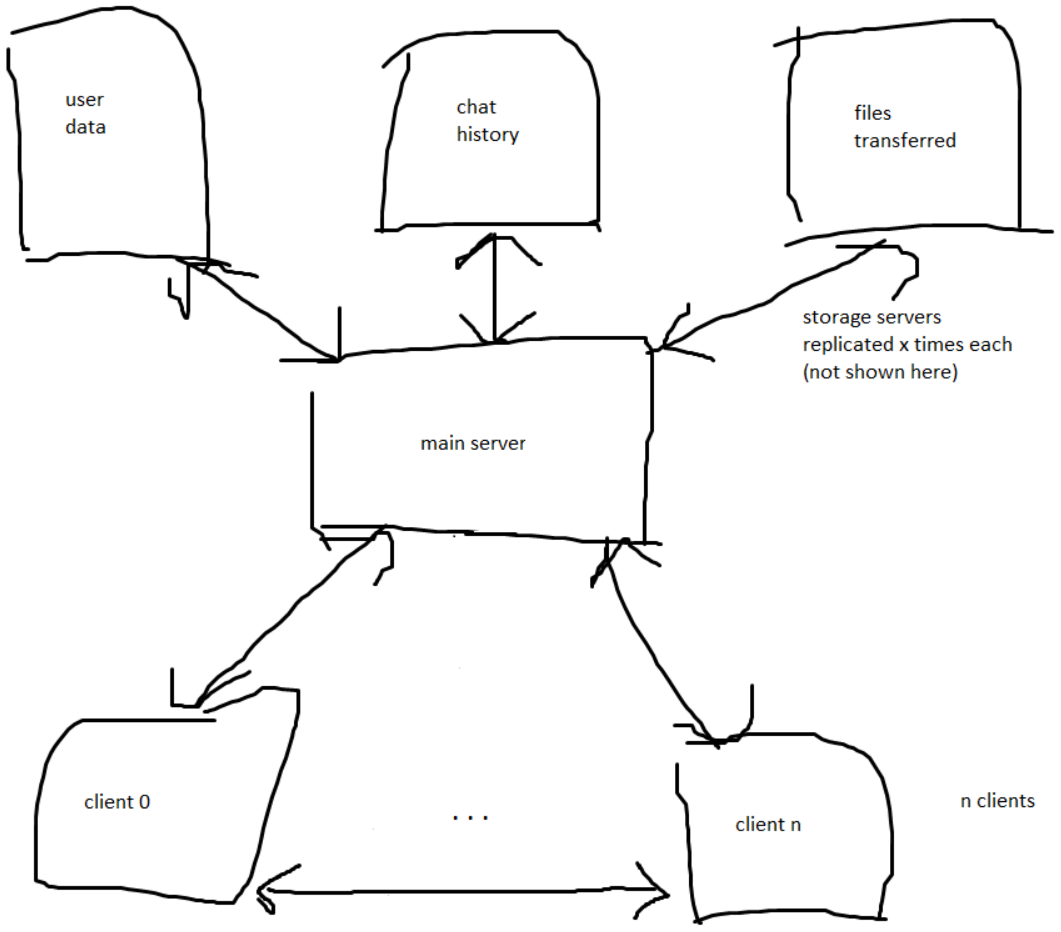
Private Message:  @username "some message"

Private file share: #share @username /path/to/file/name

Public file share: #share /path/to/file/name

Moderator: #remove username

## 4. Diagram of System



## 5. SWOT Analysis

| Strengths | Weaknesses |
|---|---|
| - Specs are similar to what we've done for previous assignments.<br>- Project is in Golang which makes it very convenient for all group members who've had a considerable amount of experience with the language. | - Project is very similar to other chat-like applications offering minimal novelty. So the project's competitiveness in the marketplace may not be as strong as the developers would like it to be.<br>- No previous use of the GoVector library or ShiViz<br>- Connecting Golang code with GUI code. |
| Opportunities | Threats |
| - Project provides an avenue to explore the benefits of various types of chat rooms. It also allows the developers to raise questions as to the design decisions of established applications. Why certain features are there, why some others are missing, etc. | - JustChat may open up various privacy issues given the 'openness' of the chat room nature. However, this threat can be minimized by file encryption, which can potentially be explored at a later date.<br>- Time commitments to do the project vary by group member |

## 6. Timeline (weeks start on Monday)

<u>Feb 29 - Mar 6</u>

System design/architecture (interface + protocols + security + clarifying scope), assign tasks

<u>Mar 7 - Mar 13</u>

Server logic, client logic, handle joining/leaving, failures, testing

<u>Mar 14 - Mar 20</u>

File transfer protocol, private messaging, testing

<u>Mar 21 - Mar 27</u>

moderator, testing

<u>Mar 28 - Apr 3</u>

chat history, testing

<u>Apr 4 - Apr 10</u>

written report, final system testing

<u>Apr 11</u>

Project Due