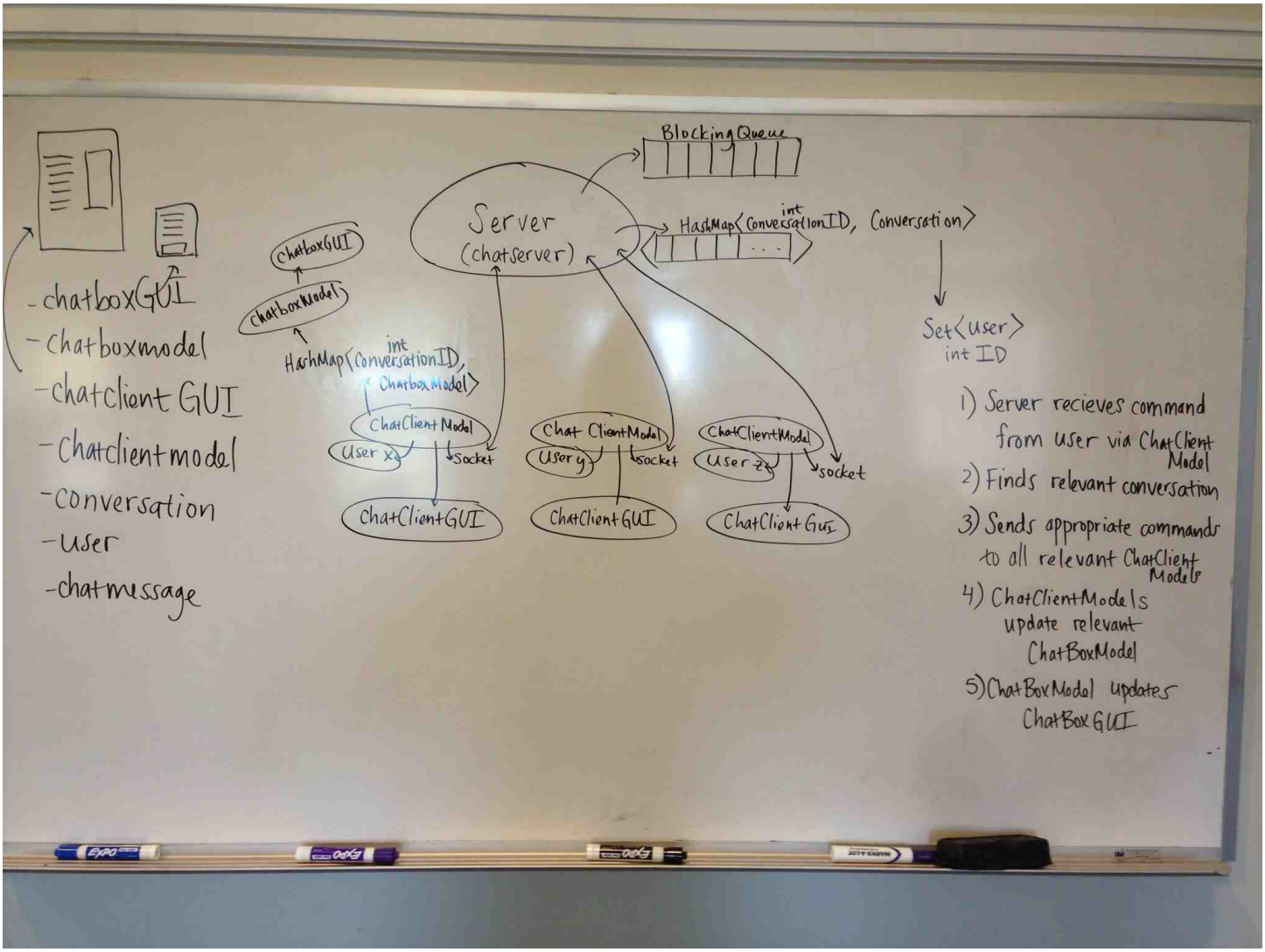
- chatboxGUI
- chatboxmodel
- chatclient GUI
- chatclientmodel
- conversation
- user
- chatmessage
ChatboxGUI
ChatboxModel
HashMap<int ConversationID, ChatboxModel>
ChatClient Model
User x
socket
ChatClientGUI
Server
(chatserver)
BlockingQueue
HashMap<int ConversationID, Conversation>
Chat ClientModel
User y
socket
ChatClientGUI
ChatClientModel
User z
socket
ChatClient Gui
Set<User>
int ID
1) Server recieves command from user via ChatClient Model
2) Finds relevant conversation
3) Sends appropriate commands to all relevant ChatClient Models
4) ChatClientModels update relevant ChatBoxModel
5) ChatBoxModel updates ChatBox GUI

Conversation Design

A conversation will be represented by a Conversation class. We will model each conversation after a chat room that must include at least one person. People will be able to join a private “chat room” by starting a chat with another person. Chats will be started with a friend by clicking on that friend’s username in the list of online people. For now, they can invite other people via a separate invite feature.

- Every conversation is initialized with two people. However, a conversation can support more people if more people are invited. Each Conversation object holds a set of User objects representing the people who are in the conversation. A conversation is terminated if zero people are in the conversation.
- The server will hold a HashMap of the current conversations. The hashmap will map conversation IDs to Conversation objects.

Conversation Class:

```
public class Conversation {

    int id;
    Set<User> users;

    /**
     * Create a new Conversation. A Conversation must
     * have at least one user.
     *
     * @param users The Users to be added to the Conversation.
     */
    public Conversation(Set<User> users, int id) {
        this.users = users;
        this.id = id;
    }

    /**
     * Add a new user to a Conversation. Do nothing if the user
     * is already part of the Conversation.
     *
     * @param user The User to be added to the Conversation.
     */
    public void addUser(User user) {
        if (!users.contains(user)) {
            users.add(user);
        }
    }
```

```
    /**
     * Remove a user from a Conversation. Do nothing if the user
     * is not part of the Conversation.
     *
     * @param user The User to be removed from the Conversation.
     */
    public void removeUser(User user) {
        if (users.contains(user)) {
            users.remove(user);
        }
    }

    public Set<User> getUsers() {
        return this.users;
    }

    public int getID() {
        return this.id;
    }

}
```

ChatMessage Class:

```
public class ChatMessage {

    private User origin;
    private String message;
    private Time time;

    /**
     * Create a new ChatMessage.
     *
     * @param origin The User from which the ChatMessage
     * originated.
     * @param message The text comprising the ChatMessage.
     */
    public ChatMessage(User origin, String message) {
        this.origin = origin;
        this.message = message;
        this.time = new Time(System.currentTimeMillis());
    }
```

```
    public User getOrigin() {
        return this.origin;
    }

    public String getMessage() {
        return this.message;
    }

    public Time getTime() {
        return new Time(time.getTime());
    }

}
```

User Class:

```
public class User {

    String username;

    /**
     * Create a new User. Each user has a String username which
     * is used to identify them.
     *
     * @param username
     */
    public User(String username) {
        this.username = username;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result
                + ((username == null) ? 0 :
username.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
```

```
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        User other = (User) obj;
        if (username == null) {
            if (other.username != null)
                return false;
        } else if (!username.equals(other.username))
            return false;
        return true;
    }

    public String getUsername() {
        return this.username;
    }

}
```

Client-Server Protocol

The basic structure of our instant messaging system will consist of a client (used through a GUI) and a server. The chat server and each chat client will send messages through a plain text protocol using basic socket communication in order to update each other on the overall state of the instant messaging system. The GUI and its controller will automatically send messages to the server and interpret messages from the server. Each message sent from the server to a client and vice-versa will be a single one-line string.

Protocol: Client to Server and Server to Client

There exists a text protocol for one-line messages from the client to the server and from the server to the client. Due to the similarity of the types of messages they will be sending to each other, we have chosen to define a single protocol for both directions of communication such that any message going in one direction is also valid in the other direction. However, different actions are performed upon receiving messages on the client side versus on the server side. The only command that is not

applicable in both directions is the “start” message, which is only useful from client to server.

Each message sent from the server to the client or from the client to the server must be of the “message” type, defined in the grammar below. Any message must be one line and cannot contain any newline characters.

In general, a username is defined to be any sequence of alphanumeric characters. An id represents a conversation id, which is unique for each conversation at any given time. An id is any positive integer. Times are represented using Unix timestamps. A line of text sent in a conversation can be anything that is one line. Our chat client allows multi-line chat messages, but they are sent over as multiple messages so that each message that represents a line of conversation only holds one line.

Protocol: Client to Server

login message: notifies the server that a new user has logged in. Sent from the client upon logging in. The server response is to notify all clients to add a new user to their list of online users using a “login” message.

logout message: notifies the server that a previously logged-in user has logged out. Sent from the client upon logging out. The server response is to notify all clients to remove this user from their list of online users using a “logout” message.

start message: notifies the server that a new conversation has begun among the specified users. When a new conversation is started by one user with another user, this message is sent to the server by the client that started the conversation. Then, the “join” message is sent to all users involved, and only at this point does the new chat box open. This is because the clients must wait for the “join” message in order to know the ID to use for the new conversation, necessary for other operations. The server response to this message is to notify all clients using the “join” message to join the specific conversation.

say message: notifies the server that within some conversation (specified with an ID), a certain line of text was sent by a specific user at some time. Sent from the client when a new line of conversation is entered into a conversation. The server response to this message is to notify all clients using the “say” message to add the new line of chat to the specified conversation.

join message: notifies the server that a new user has joined the specified conversation. Sent from the client when a new user is invited to join a conversation. In order for the new user to actually join, the server must send a “join” message to that client with the specified conversation ID. The server response to this message is to add the user to the list of users in the conversation and then notify all clients that a new user has joined the conversation using the “join” message.

leave message: notifies the server that a user has left the specified conversation. Sent from the client when the user leaves a conversation. The server response is to remove the user from the list of users in the conversation and then notify all clients that a user has left the conversation using the “leave” message.

Protocol: Server to Client

login message: notifies the client that a new user has logged in. The client response is to add this new user to the list of online users.

logout message: notifies the client that a user has logged out. The client response is to remove this user from the list of online users.

start message: not used.

say message: notifies the client that a new line of text from a specific user should be added to a conversation with the given timestamp. The client response is to update the correct chat box with the new line of text.

join message: notifies the client that a new user has joined a conversation. The client response is to update the correct chat box with a message saying that the new user has joined.

leave message: notifies the client that a user has left a conversation. The client response is to update the correct chat box with a message saying that the user has left the conversation.

Protocol Grammar

```
message :== login | logout | start | say | join | leave
login :== “login” SPACE username
logout :== “logout” SPACE username
start :== “start” (SPACE username)+
say :== “say” SPACE id SPACE username SPACE time SPACE text
join :== “join” SPACE id SPACE username
leave :== “leave” SPACE id SPACE username
username :== [0-9A-Za-z]+
id :== [1-9][0-9]*
time :== [1-9][0-9]*
text :== [^\r\n]*
```

Catching Username Conflicts

When a new client is started, it will ask the user for a username. This username is then sent to the server using the “login” message. If the username is valid, the server will reserve it and then send the “login” message to all clients that are already with usernames, as well as the newly connected client. Else, the username will immediately send back the “logout” message to the specific client that requested the bad username.

State: Server

At any point, the server will know its socket, the sockets (and usernames) of all currently connected clients, and the list of ongoing conversations. In addition, it will run a thread for each client connected. A conversation consists of a list of users and a unique ID number.

```
public class ChatServer
```

- `// this class serves as the chat server`
- `private ServerSocket socket;`
  - `Used to communicate with clients.`
- `private HashMap<User, Socket> clients;`
  - `Holds a list of all connected clients.`
  - `Updated whenever a user logs in or logs out.`
- `private HashMap<Integer, Conversation> conversations;`
  - `Integer corresponds to conversation ID.`
  - `How do we assign conversation IDs? Assign smallest possible positive integer that is not currently used.`

    - Updated whenever a new conversation begins, or when a conversation ends (when all participating users leave).
- private BlockingQueue<String> queue;
    - Stores messages from clients that run() uses to make changes to the current state and send messages back to clients.
- public ChatServer(ServerSocket socket);
    - Creates a new chat server with some socket for clients to connect to.
    - Creates a new thread to process messages using the run() method.
- public void serve();
    - Sets up the server socket and accepts connections, delegating a new thread for each new connection (using a ChatServerClientThread).
- private void sendMessageToClients(String message, List<User> clients);
    - Sends the message to all the clients listed, with help from the HashMap clients.
- private void work();
    - Run in its own thread to process messages from its BlockingQueue as they come in.
    - Updates instance variables accordingly.
- public void addMessageToQueue();
    - Used by ChatServerClientThread to add a message to its queue.

public class ChatServerClientThread implements Runnable

- // A thread run for every client (username) connected to the server.
- private Socket mySocket;
- private ChatServer myServer;
    - Updated based on messages received from the client.
- public void ChatServerClientThread(Socket socket, ChatServer server);
    - Creates a new ChatServerClientThread with the given socket and associated server.
- public void run()
    - Reads messages from the client socket and has the server respond accordingly using thread-safe methods.
    - Adds messages to server's message queue.

State: Client

At any point, the client's state holds its current socket, a list of online users, as well as a list of ongoing conversations. A conversation consists of a list of users and a unique ID number. The client consists of the models for the GUI elements.

Both models are ActionListeners, meaning that they will react to things that happen in the GUI.

```
public class ChatClientModel implements ActionListener
```

- // The model that communicates with the server and updates the chat client GUI.
- private Socket socket;
  - Used to communicate with the server.
- private HashSet<User> users;
  - List of online users.
- private HashMap<Integer, ChatBoxModel> conversations;
  - IDs of conversations mapped to the ChatBoxModels corresponding to the conversations.
- private ChatClient myGUI;
  - The GUI that is updated by this model.
- public ChatClientModel(ChatClient gui, Socket socket);
  - Creates a new ChatClientModel with the specified GUI and socket.
- public void work();
  - Waits for server messages and responds accordingly by either updating its GUI or by notifying one of its ChatBoxModels.
- public void actionPerformed(ActionEvent e);
  - Sends a message to the server in response to an ActionEvent from the GUI.
- public void sendMessageToServer(String message);
  - Sends the message to the server.

```
public class ChatBoxModel implements ActionListener
```

- // One of these is created for each new window that is a chat box.
- Conversation conversation;
  - Holds the current conversation.
- ChatBox myGUI;
  - Corresponds to the chat box that is updated by this model.
- ChatClientModel myClient;
- public ChatBoxModel(ChatBox gui, ChatClientModel client);

    - Creates a new ChatBoxModel with the specified GUI, the specified client, and an empty conversation.
- public void addUserToConversation(User user);
- public void removeUserFromConversation(User user);
- public void addLineToConversation(ChatMessage message);
- public void endConversation();
- public void actionPerformed(ActionEvent e);
    - Sends a message to the server (through the client) in response to an ActionEvent from the GUI.