Conversation Definition

Conversations will be defined much like chat rooms so that users can enter and exit freely. This provides flexibility so that an arbitrary number of users can be the in the same conversation. Thus, there will be no distinction between a chat between k > 2 people and a chat between 2 people. A chat room will close as soon as all of the members of a chat have left.

In order to implement conversations as chat rooms, we must make each conversation distinct. Each conversation will be uniquely identified by an integer called the ChatId, which is an integer between 1 and 2^32. When a client wants to begin a conversation with another client, he will send a “create” command to the server with his UserId and the other client’s UserId. The server responds by creating a chat room with a new ChatId not currently in use, and sends a response back to both users. The server will then hold an instance of a conversation object, which will periodically be updated by requests sent from the clients. Whenever the conversation is updated, the server will send update to each of the clients that is still connected to the conversation.

When a client wants to connect to a conversation, he will write a join command, defined in the client-server protocol, and specify a ChatId. The ChatId tells the server which conversation the user would like to join. When a client wants to exit a conversation, he will write an exit command, which is also defined in the client-server protocol. The ChatId will be included in this command, and will allow the server to disconnect the user from the conversation. If the user’s exit causes the conversation to be empty, the server will delete the conversation object and recycle the ChatId.

Specifically, a Conversation class will be created on the server end, and a ConversationClone class which will be updated by the server whenever the actual conversation is updated. The Conversation class will have the following public methods: addText(), addUser(), deleteUser(), getCurrentUsers(), getHistory(). The class will have a private list of all the messages that have previously been sent to the conversation called history. Also, the class will have a list of current users called currentusers.

The addText() method takes a UserId and a string as input, then adds the new text into the conversation history. The addUser() method takes a UserId and adds it to the currentUsers list in the conversation. The deleteUser() method deletes a given UserId from the list of current users in the conversation. The getCurrentUsers() method returns a deep copy of the UserIds that are currently part of the conversation. The getHistory() method returns the the messages (and their users) sent during the conversation in list format.

Next, a ConversationClone class will have an update(), getCurrentUsers(), and getHistory() method and will have a list of previous messages and their users in the history list, analogous to the history list in the Conversation class. Moreover, the ConversationClone class will contain a list of users called currentUsers. The update() method changes the currentUsers and the history list in the ConversationClone class. The getCurrentUsers() and getHistory() methods are analogous to their counterparts in the Conversation class and return a list of users and a list of messages respectively.