guichat: Preliminary Design

Note on Scale

The fact that our chat server is intended for small-scale use has several consequences in terms of design. First, we will be running the server as a single process with multiple threads. Second, **we will not protect our server and client from malicious network users**. Our product is specified to work if all users are running the client we provide with the server we provide, but we make no guarantees about the behavior of either if there is a rogue user sending bad messages around on telnet.

Conversations

Description

We define a conversation as a connection between any positive number of users who are connected to the IM server. Any user connected to the server can

- create a conversation, assigning the conversation a name which is distinct from all already-existing conversations.
- connect to a conversation by specifying its name, if he is not already in that conversation.

Initially, a conversation contains only the user who created it. Upon creation, the user may specify a unique name for the conversation or have a unique name automatically generated by the server. Any user in a given conversation can

- send a message, which will be seen by all other users in the conversation
- invite another user to the conversation
- leave the conversation.

When user A invites user B to join a conversation, user B’s client will allow him to accept or reject the invitation. If he accepts, the client will join the room in the same way that user A joined the room.

Implementation

Conversations are represented by instance of Conversation. The Conversation class has the following instance variables:

- users
  A set of Users that are currently in the conversation. For this field, an iteration-safe data type will be necessary. One feasible option is CopyOnWriteArraySet. This will not be a performance bottleneck because the rate at which users enter and leave the room is on the scale of network operations, which are much slower than memory or processor operations.
- name
  The name of the conversation represented by this Conversation (a string). This string will be globally unique among all conversations on the server.

Conversation contains the following methods, all of which are called by the associated methods in IMServer. For more detailed descriptions on usage, see the corresponding descriptions of methods in the Server section below.

- sendMessage(User u, String m, int uniqueID)
  Sends message m to each User in this.users with sender name u and ID uniqueID.
- add(User u)
  If this.users does not already contain u, adds u to this.users and sends an added to conversation message to every client in this conversation, including to u.
- remove(User u)
  If this.users contains u, removes u from this.users and sends a removed from conversation message to every client in this conversation.

Server

Description
The server has a main thread that listens to a specific socket, corresponding to a specific port and IP address, for user connections. The server also keeps a thread for each client, which handles the network requests from that client.

Implementation
The server is implemented in the IMServer class, which implements Runnable. Instances of IMServer keep track of the following instance variables:

- users
  A set of Users that are currently connected to the server.
- conversations
  A map from Strings to Conversations that keeps track of active conversations. The key for a given conversation is the conversation’s name.

IMServer also contains the following methods, which are called from the run method of

User when network messages are received from a client:

- sendMessage(User u, String convName, int messageId, String m)
  Sends the message m to all clients in the conversation specified by convName with sender name u and ID messageId. The variable messageId should be unique among all messages sent by this client to the conversation associated with convName. (In practice, this number will be globally unique among *all* messages this client has sent, since this is easier to implement.) Note: since messages will be logged on the client side, there is no need for the server to store messages at all.
- newConversation(User u, String convName)
  If convName is non-null and non-empty and there is no Conversation associated with convName in this.conversations, creates a new Conversation with name convName containing only u and adds it to this.conversations. If convName is null or empty, creates a new Conversation associated with a unique String containing only u and adds it to this.conversations. Sends a new conversation receipt message to the client represented by u with information on whether or not the conversation was successfully created. newConversation is thread-safe because it synchronizes on conversations to prevent multiple users from creating new conversations, which could possibly have the same name, at the same time.
- addToConversation(String convName, User u)
  If there is a Conversation associated with convName, adds u to that Conversation and sends an added to conversation message to all Users in it, including to u. If there is no Conversation associated with convName, sends a removed from conversation message to the client associated with u.
- removeFromConversation(User u, String convName)
  If there is a Conversation associated with convName, removes u from the Conversation, sends a removed from conversation message to every other User in the Conversation, and removes the Conversation from this.conversations if the Conversation contains no Users after removing u. Otherwise, does nothing.
- disconnectUser(User u)
  Removes User u from each of his conversations, sending a removed from conversation message to every other user in the conversation, using exitConversation, and removing empty conversations from this.conversations. Removes u from this.users. Sends a disconnected message to all clients.

We use the publish-subscribe pattern. When a user enters, leaves, or sends a message to a conversation, we fire a notification to all the other users in the conversation, which allows their clients to update the displayed list of participants or message history.

The server also contains a method run which makes the server listen for user connections on the calling thread. The private method connectUser, which adds a new User object to this.users, starts it, and sends a connected message to all clients, is called from this thread whenever the server receives a connection request.

User

Description
Instances of User are used by the server to keep track of connections to clients. Note that we frequently use User to refer to the client represented by an instance of User.

Implementation
User is a subclass of Thread and has the following instance variables:

- socket
  The socket over which the server communicates with this client.
- username
  A String containing the client's username. username is non-null, non-empty, contains at most 256 characters, and contains no new-line characters.
- conversations
  A set of Strings representing the names of all the conversations that this client is in.

The run method of User listens to the socket, parses network messages when they are received, and calls the corresponding Server method.

Network Protocol

There are seven types of network messages sent from the client to the server:

1. CONNECT
   A connect message is sent as a login request to the server.
2. IM
   An IM message is sent as a request to send a given message to all users in a given conversation.
3. NEW_CONV
   A new conversation message is sent as a request to create a new conversation. The user may specify a name to request a conversation with that name.
4. ADD_TO_CONV
   An add to conversation message is sent as a request to add a given user to a given conversation.
5. ENTER_CONV
   An enter conversation message is sent as a request to be added to a given

conversation.
6. EXIT_CONV
   An enter conversation message is sent as a request to exit a given conversation.
7. DISCONNECT
   A disconnect message is sent as a logoff request from the server.

The client-to-server network protocol grammar is as follows:

```
MESSAGE = MESSAGE_TYPE NEWLINE MESSAGE_CONTENT
MESSAGE_TYPE = CONNECT | IM | NEW_CONV | ADD_TO_CONV | ENTER_CONV |
        EXIT_CONV | DISCONNECT
MESSAGE_CONTENT = CONNECT_CONTENT | IM_CONTENT | NEW_CONV_CONTENT
        | ADD_TO_CONV_CONTENT | ENTER_CONV_CONTENT | EXIT_CONV_CONTENT
        | DISCONNECT_CONTENT

CONNECT_CONTENT = USERNAME
IM_CONTENT = CONV_NAME NEWLINE IM_ID NEWLINE MESSAGE_TEXT
NEW_CONV_CONTENT = NEW_CONV_NAME
ADD_TO_CONV_CONTENT = USERNAME NEWLINE CONV_NAME
ENTER_CONV_CONTENT = CONV_NAME
EXIT_CONV_CONTENT = CONV_NAME
DISCONNECT_CONTENT = NOTHING

USERNAME = [^\n]{1,256}
NEW_CONV_NAME = [^\n]{0,256}
CONV_NAME = [^\n]{1,256}
IM_ID = [0-9]{1,10}
MESSAGE_TEXT = [^\n]{1,512}
NEWLINE = \n
NOTHING = .{0}

CONNECT = 0
IM = 1
NEW_CONV = 2
ADD_TO_CONV = 3
ENTER_CONV = 4
EXIT_CONV = 5
DISCONNECT = 6
```

There are seven types of network messages sent from the server to the client:

1. INIT_USERS_LIST
   An initial users list message is sent upon connection with the server. The message specifies all other users that are currently logged in.
2. IM
   An IM message is used to send a message to all users in a conversation.
3. NEW_CONV_RECEIPT
   A new conversation receipt message confirms that a given conversation was properly created.
4. ADDED_TO_CONV
   An added to conversation message notifies the client that a user has been added to a conversation that the client is in.
5. REMOVED_FROM_CONV
   A removed from conversation message notifies the client either that a user has been removed from a given conversation or that the server failed to add the client to the given conversation.
6. CONNECTED
   A connected message notifies the client that a new user has connected to the server.
7. DISCONNECTED
   A disconnected message notifies the client that a given user has disconnected from the server.

The server-to-client network protocol grammar is as follows. Note that some pieces of the grammar may be different from their counterparts with the same name in the grammar for the client-to-server network protocol grammar; for example, IM_CONTENT in this grammar contains the sending user's username:

```
MESSAGE = MESSAGE_TYPE NEWLINE MESSAGE_CONTENT
MESSAGE_TYPE = INIT_USERS_LIST | IM | NEW_CONV_RECEIPT | ADDED_TO_CONV
        | REMOVED_FROM_CONV | CONNECTED | DISCONNECTED
MESSAGE_CONTENT = INIT_USERS_LIST_CONTENT | IM_CONTENT |
        NEW_CONV_RECEIPT_CONTENT | ADDED_TO_CONV_CONTENT |
        REMOVED_FROM_CONV_CONTENT | CONNECTED_CONTENT |
        DISCONNECTED_CONTENT

INIT_USERS_LIST_CONTENT = (USERNAME NEWLINE)*
IM_CONTENT = USERNAME NEWLINE CONV_NAME NEWLINE IM_ID NEWLINE
        MESSAGE_TEXT
NEW_CONV_RECEIPT_CONTENT = (SUCCESS | FAILURE) NEWLINE CONV_NAME
ADDED_TO_CONV_CONTENT = USERNAME NEWLINE CONV_NAME
REMOVED_FROM_CONV_CONTENT = USERNAME NEWLINE CONV_NAME
```

```
CONNECTED_CONTENT = USERNAME
DISCONNECTED_CONTENT = USERNAME

USERNAME = [^\n]{1,256}
CONV_NAME = [^\n]{256}
IM_ID = [0-9]{1,10}
MESSAGE_TEXT = [^\n]{1,512}
NEWLINE = \n

INIT_USERS_LIST = 0
IM = 1
NEW_CONV_RECEIPT = 2
ADDED_TO_CONV = 3
REMOVED_FROM_CONV = 4
CONNECTED = 5
DISCONNECTED = 6
```