GUI Chat Design Documents

Colleen Josephson, Kiranmayi Bhattaram, Michael Ahearn

Table of Contents

Functionality Overview

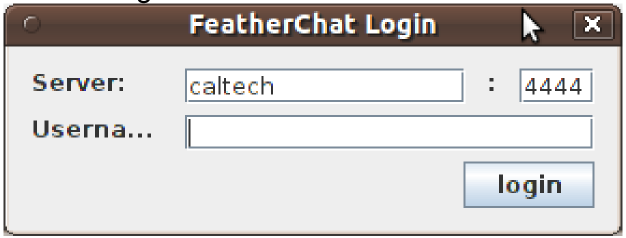
Conversation Design

The conversation design follows a publish-subscribe pattern and is inspired by IRC's notion of "channels". A conversation can be either **public** or **private**. Each conversation is associated with a list of users who can participate. If this list is empty, the conversation is public, but otherwise, it is private to the users on the list, and only they are made aware of its existence. A conversation functions like a chat room: anyone who joins the conversation receives all messages sent to the conversation. Once a conversation has no participants, it is deleted (this is to prevent the accumulation of dead conversations from wasting server space). Any user logged into the server can create a conversation.

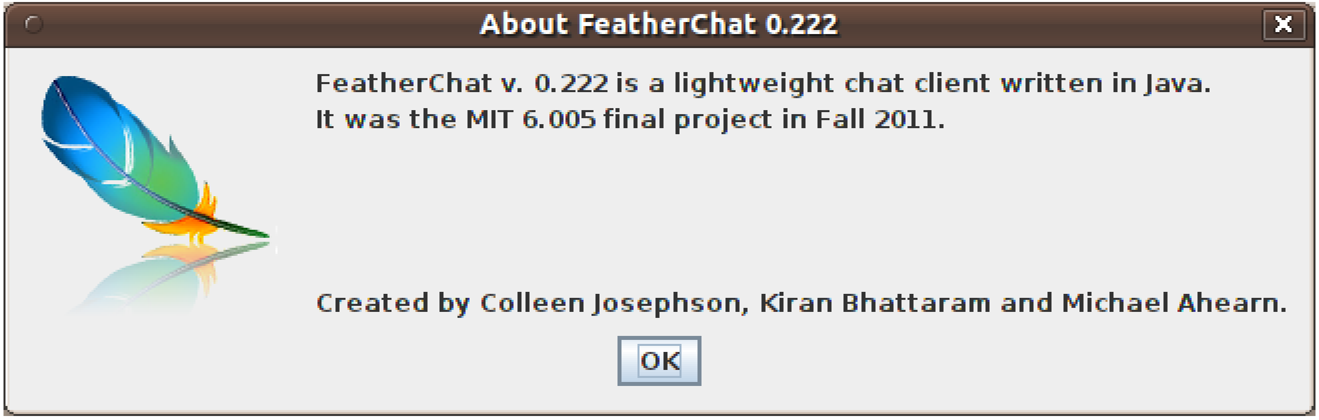
A **private message** is basically implemented as a private conversation with 2 allowed users. The name of the conversation is defined as "user1_to_user2", where the user names are ordered lexicographically. Any user can invite another user to a private conversation; doing so pops up a conversation window on both ends.

GUI Design and Look

The GUI design we agreed upon is clean and minimalist. The first screen is a login screen, which takes you to a main window. From here, the user can immediately see existing public conversations and currently logged-in users. A conversation window is created for every new conversation, split into messages and user states.

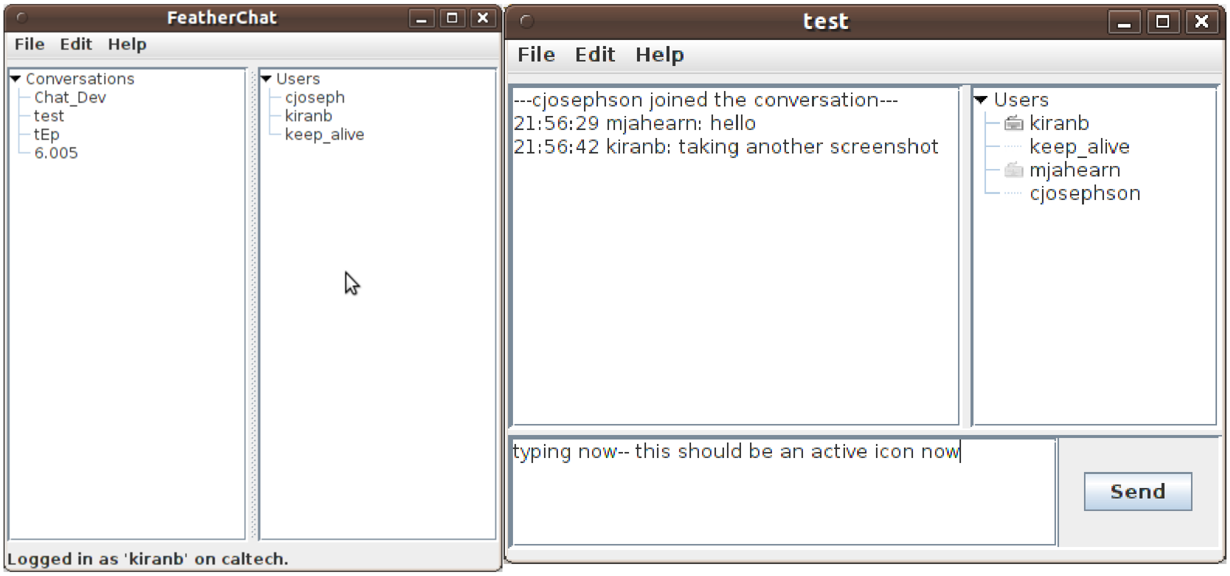


Server: caltech : 4444
Userna...



FeatherChat v. 0.222 is a lightweight chat client written in Java.
It was the MIT 6.005 final project in Fall 2011.

Created by Colleen Josephson, Kiran Bhattaram and Michael Ahearn.



```
---cjosephson joined the conversation---
21:56:29 mjahearn: hello
21:56:42 kiranb: taking another screenshot
```

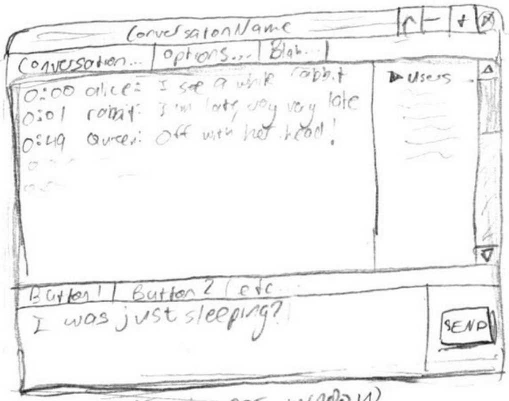
typing now-- this should be an active icon now

Logged in as 'kiranb' on caltech.

Users' states are implemented by icons: a keyboard implies typing, a greyed out keyboard implies the user has entered text, and three dots implies no text has been entered. In the above screenshot, for example, kiranb is currently typing, mjahearn has entered text, and the other 2 users are idle.

Double-clicking on a conversation joins it, and double-clicking on a user starts a p2p conversation. The user can create new conversations or join any conversation through the *File* menu. The act of creating a person-to-person conversations pops up a chat box on the receiving end. Typing, however, does not bring up a new box if the other party decides to leave. A user can re-join a p2p conversation once they have left it. The message history, however, will have been cleared.

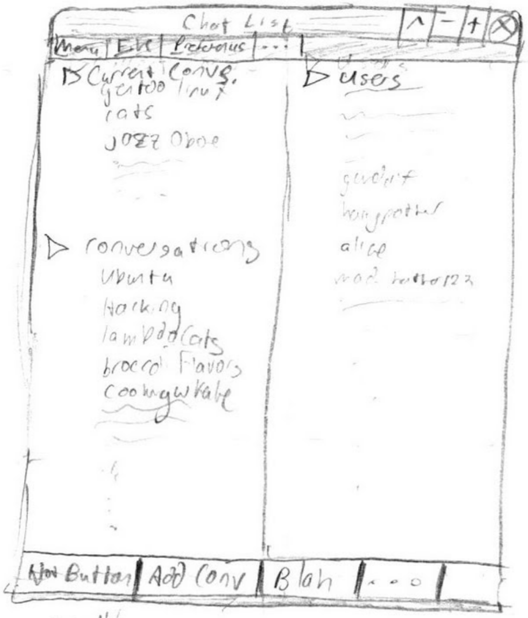
Original Sketches



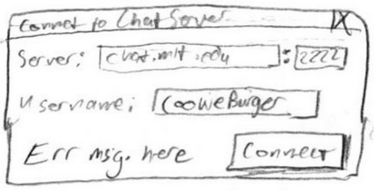
0:00 alice: I see a white rabbit
0:01 rabbit: I'm late, very very late
0:49 Queen: Off with her head!

I was just sleeping?

- one conversation per window
  - tabs a potential 2nd poss
- 'enter' key sends msg, as well as send button
- 'midbar' buttons changed based on whether you are a moderator of the conv
  - don't make add/remove allowed user options visible to non-privileged
  - if moderator status changes mid-convo, GUI does not reflect changes until window re-opened



- scrollbars appear when necessary
- user/conversation lists collapsable



Server: chat.mit.edu : 2222

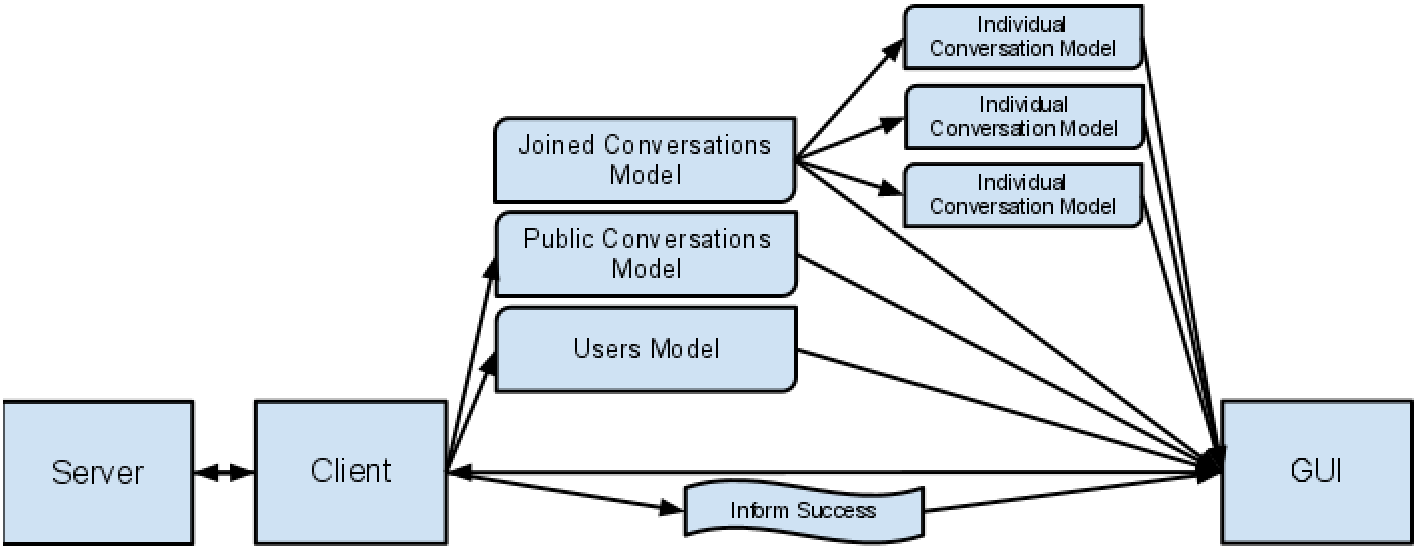
Username: CookieBurger

Err msg. here

- 2nd pass: remember last-used server & username

Software Design

Information and Control Flow


Individual Conversation Model
Individual Conversation Model
Individual Conversation Model
Joined Conversations Model
Public Conversations Model
Users Model
Server
Client
Inform Success
GUI

Arrows in this diagram represent information and control flow.

The server and client communicate through a text-based grammar (outlined later) over sockets, following the traditional Client-Server model.

The Client and GUI interact through a modified MVC model, which relies on listeners and callbacks.
There are 2 sorts of interactions between the client and the GUI. In the first, the GUI originates a request that requires a response. In the second, the client receives updates from the server, and must notify the GUI of the change.

- Control flow for a server update:
  - The client processes the server update and updates the appropriate model.
  - The GUI is registered as an Observer on the Observable model. Thus, when the model changes, the GUI is notified, and changes accordingly.
  - For example, if a user joins a conversation, the server will send an UPDATE-CONVUSERS message. The client will parse this, and then inform all the registered listeners. The listener will then add the new user to the list in the GUI.
- Control flow for a request:
  - The GUI calls the appropriate method on the client, and provides the client with and InformSuccess object, which has fail and succeed methods that the GUI defines.
  - The client processes the request, and sends it to the server.
  - When the server responds, the client processes the response and calls the appropriate method (fail or succeed) on the associated InformSuccess object. The client also changes the associated models, if necessary.
  - The fail/success method (and the model, if applicable) updates the GUI.

Server Design

Datatypes:

- ConversationMaker, a Conversation factory
- Conversation (Interface)
- RealConversation (implements Conversation)
- UaerMaker, a User factory
- User (Interface)
- RealUser (implements Conversation)

The server has a main listen loop that listens for incoming connections on the ServerSocket. A new socket is then created, and a thread is started to listen for incoming requests. Each socket has it's own thread.

When an incoming request is read from the socket, the server hands it to handleRequest(), which parses the request appropriately. The parsing method is very simple. The request is split by spaces. If you look at the grammar definition, each request starts with a request type, e.g. JOIN, LEAVE, LIST-USERS. A hash map is defined at the top of ChatServer.java that maps each possible request type to a Function object (as described in lecture 15). The parser then applies the method to the string. Each little function does basic error checking, and then forwards the information worker functions, which do the appropriate action and return the response.

The possible responses are mostly defined in the ACK enum, making the grammar easy to see and work with. The use of functional programming style here is ideal for such a simple protocol, making parsing simple and effective.

The server uses two main types of datatypes, Conversations and Users. The User object contains the username and the socket. The conversation object contains all the Users in a conversation, and other useful information, such as the list of users allowed to enter a private conversation. The server is instantiated with a ConversationMaker and a UserMaker factory, making the server more robust and extensible. The run() method of the server, in addition to starting the main handleConnection() loop, also makes a cleanup thread that periodically polls all users and removes those who have disconnected without logging off. This thread also deletes all Conversations with 0 users.

Most requests merely require a response, but some are more complicated. For example, messaging requires sending information to many users. The server passes the responisibility of relaying messages over to the Conversation object by calling .publish(). This forwards the message to all Users currently in the channel. A similar method, .publishUpdates() is used to send updates about user typing state (has_typed, is_typing, no_text) and other channel data (e.g. updating the users list if a person leaves the conversation).

Client/GUI Design

Client Datatypes:

- ConversationModel, a model of all the *public* conversations on the server
- UsersModel, a model of all the users on the server
- JoinedConversationsModel, a model of all the conversations the user is in, both public and private
- FailureModel, an observable to tell the GUI when the server is suddenly

unreachable.

- Evaluate, an interface
- Evaluator, implements evaluate

GUI Datatypes:

- ChatGUI, the main GUI window
- 7ChatWindow, conversation windows
- Logon, the login prompt
- CreateConversation, the new conversation prompt
- ChatFrame, extends JFrame. Children: ChatWindow and ChatGUI
- ChatDialog, extends JDialog. Children: Logon and CreateConversation
- Custom JComponents to improve the look and feel

We make extensive use of the listener pattern and callback methods. Every request that requires a response to the server is submitted with an InformSuccess object. This contains .success() and .fail() methods. If the client deems the input invalid (e.g. usrename has a space), or if the server returns and error response, .fail() is called; .success() is called otherwise. Since references to the location it was created in survive, these InfomeSucces objects allow the client to call GUI methods without breaking an abstraction barrier.

When the client receives a request from the GUI, it creates an Evaluator object, which deals with evaluating the success of the server response. The evaluator also deals with any client-side changes that need to be made based on the server response. The GUI also presents the method with an InformSuccess object, which implements succeed and fail methods. The client then creates a new message ID for this request, and places the Evaluator and InformSuccess methods in a map, keyed by the message ID. Finally, it sends out the message to the server.

The client has a separate listen() thread that checks for server messages, and deals with them appropriately. If the message is a response to a request, the client looks up the message ID in the map populated earlier, uses the Evaluator to parse the message. Based on the Evaluator's result, the client then calls the appropriate method (succeed or fail) on the InformSuccess.

The GUI can also register Observer objects with certain models in the client, which is an implementation of the listener pattern. For example, we have a ConversationsModel that notifies the GUI when a conversation has been created or destroyed, causing the Conversations list to be updated accordingly. The main GUI registers listeners in it's constructor. Each window also registers a listener to be notified of incoming messages.

Concurrency Strategy

Server Concurrency

The server makes extensive use of synchronization. All methods that deal with shared data are synchronized. The constructors are run only once, so do not need to be synchronized. The main the main run method only accesses the final variable port, so it does not need to be synchronized. The run() method does start other threads, but those threads do not directly access instance data. All access to instance data is made in calls to synchronized methods.

Client/GUI Concurrency

The Client only runs one thread itself: the listen() thread. This parses the server messages and either updates the models (all thread-safe) or calls the Evaluate and InformSuccess methods (also thread-safe after creation).

Creating message IDs is implemented by incrementing and resetting an AtomicInteger, which is done from a thread that requires a lock on the AtomicInteger.

The GUI starts a new thread for all client methods that require a submit to the server so that the GUI stays responsive while waiting for a server response. Server communication threads and callback methods surround GUI updates with an invokeLater() so that they are added to the Swing queue.

General Concurrency

Ordering of conversation messages is dealt with server-side. Timestamps are added to messages as the server receives them.

Testing Strategy

Server Testing

All the basic functionality of the server is tested, including failure conditions.

- A method called testHandleRequest(String msg) passes a message to the server's parser, and returns a String array response.
- Since handleRequest() passes MSG sending responsibility to Conversations, we had to make dummy Conversation/User objects so we can see which messages each channel/ user receive.

Testing coverage:

1. First, make sure that the server functions properly under correct use cases:
    a. Test that joining and leaving the server are handled correctly.
        i. correct responses
        ii. users.size == 1
    b. Test that each action unrelated to a specific conversation works when logged into a server
        i. list users, list conversations, etc.
    c. Test that each action related to a specific conversation works while in one or multiple conversations
        i. join/leave
        ii. listConvUsers
        iii. MSG
        iv. update user states
        v. channel deletion cycle
2. Then, make sure that the server handles abnormal use cases properly:
    a. Test that the server returns an error for any kind of improperly formatted input.
    b. Test that the server denies attempts by a user to join a conversation without permission.
    c. Test that the server returns an error (but takes no other action) when a user attempts to join a server or conversation that they have already joined or leave a server or conversation that they have already left.
    d. Test that the server denies attempts to join nonexistent conversations.

Client Testing

Again, all the basic functionality is tested.

- Use public method handle(String s) to directly see how the client responds to a request.
- Use the Unit test to create a middleman that the client can write to (a TestObject)
- Create testing InformSuccess objects that call Assert.fail(String message) if the wrong success method is called (fail called when succeed is expected, or vice versa)

Test coverage:

1. Test expected use cases:
    a. Test that joining and leaving the server are handled correctly.
    b. Test that each action unrelated to a specific conversation works when logged on to a server.

    i. *requests*: list users, list convs, get username, etc.
    ii. *updates:* update users, update conversations, etc.

c. Test that all actions related to a specific conversation works while in one or multiple conversations.
    i. *requests*: join/leave conversations, listConvUsers
    ii. *broadcasts*: MSG, user state updates
    iii. *server updates*: NEW messages, other user state updates, conversation user updates

2. Then, make sure that the client handles error messages properly:
    a. Test that creating a new conversation calls the InformSuccess.fail() method if the server returns that the name is duplicated.
    b. Test that joining a conversation calls the InformSuccess.fail() method if the server returns that the user is already part of the conversation.
    c. Test that leaving a conversation calls the InformSuccess.fail() method if the server returns that the user is not a part of the conversation.
    d. Test that joining a conversation calls the .fail() method if the server returns that such a conversation does not exist.
    e. Test that joining a private conversation calls the .fail() method if the server returns that the client does not have the appropriate permissions.

GUI and End-to-End Testing

We tested the GUI thoroughly by going through a checklist of actions (included in the end) after every major commit, in addition to dogfooding.

Once we had a functional product, Colleen set up a constantly running server. We moved most team communication to the server, and Colleen invited her living group to test out the system for a few days. Thus, we tested the server with a large number of users connected, and across multiple operating systems (Windows, Mac, and various flavors of Linux).

Test Coverage:

- Logging in
    - logging in to a server with a large latency shouldn't affect GUI responsivity
    - error messages (couldn't find host, invalid username, username exists, etc.) should display correctly
- Creating a new public conversation
    - public conversation list in main window should update
    - users in conversation window should display correctly
    - creating a duplicate conversation should pop up the "Conversation exists" error window
- Creating a new private conversation
    - users in conversation window should display correctly
    - user not in allowed list should not be able to join the conversation
    - creating a duplicate conversation should pop up the "Conversation exists" error window
- Creating a new p2p conversation
    - users in conversation should display correctly
    - a new conversation window should pop up on both ends of the conversation
    - a user should be able to close the conversation window and rejoin it later
- Joining a conversation
    - users in conversation should display correctly
    - users in conversation should update correctly for all other users
    - attempting to join a nonexistent conversation should pop up the "Nonexistent conversation" error window
- General conversation checks
    - check that all the user states (is_typing, has_typed, and no_text) display as

expected

- check that adding messages works as expected (user types, and the message shows up when it gets back from the server, other users receive the message, etc.)
- Test that all actions perform to specification when performed in an arbitrary, valid order. (sending many messages, adding/removing a # of users from a channel, creating/removing channels, etc.)

Design Updates

A few changes were made to our original design.

- The idea that client requests to the server block on a server response was scrapped. Instead, we implemented the Listener pattern and callbacks, and developed the client and the GUI in parallel.
    - This involved creating Observable Models and InformSuccess objects that allowed the client to send information to the GUI.
- The server now sends out updated user, conversation, and conversation user lists every time a change happens.
- Implementing administrators for private conversations was deemed to be unecessary, and was scrapped.

Grammar

terminals:

- sp := “ “
- msg_ID := [0-9]+
- Username := [A-Za-z0-9_.-]
- Conversation := [^ whitespace]
- actual_message := [^ ‘\n’]
- state := “is_typing” || “has_typed” || “no_text”

Client-->Server

Message Purpose	Message
Connect to server	“HELLO” sp msg_ID sp Username EOM
Leave server	“BYE” sp msg_ID sp Username EOM
Create conversation	“CREATE” sp msg_ID sp Username Conversation (Username)* EOM
Join conversation	“JOIN” sp msg_ID sp Username sp Conversation EOM
Leave conversation	“LEAVE” sp msg_ID sp Username sp Conversation EOM
Message conversation	“MSG” sp msg_ID sp Username sp Conversation sp actual_message EOM
List users connected	“LIST-USERS” sp msg_ID EOM
List users in conversation	“LIST-CONVUSERS” sp msg_ID sp Conversation EOM
List public conversations	“LIST-CONVS” sp msg_ID EOM
Invite user to private msg	“INVITE” sp msg_ID sp Username sp Conversation EOM
Update user state	“UPDATE-USERSTATE” sp msg_ID sp Username sp Conversation sp State EOM

Server-->Client

Message Purpose	Message
Malformed Request Error, no MsgID	ERROR MALFORMED EOM
Malformed Request Error, msgID available	ERROR sp msg_ID sp MALFORMED EOM
General Error	ERROR sp msg_ID EOM

Successful login	HELLO sp msg_ID sp Username EOM
Username taken	ERROR sp msg_ID sp USERNAME-TAKEN sp Username EOM
Successful logoff	BYE sp msg_ID sp Username EOM
Successful join	JOIN sp msg_ID sp Username sp Conversation EOM
No such conversation	ERROR sp msg_ID sp CONV-NONEXIST sp Conversation EOM
Lack permission to join conversation	ERROR sp msg_ID sp CONV-JOIN-PERM sp Comversation EOM
Already joined	ERROR sp msg_ID sp USER-IN-CONV sp Username cp Conversation EOM
Successful leave	LEAVE sp msg_ID ssp Username sp Conversation EOM
Already left	ERROR sp msg_ID sp USER-NOT-IN-CONV sp Conversation EOM
Successful create	CREATE sp msg_ID sp Conversation EOM
Conversation exists	ERROR sp msg_ID sp CONV-EXIST Conversation EOM
No Such User	ERROR sp msg_ID sp
Update user state	UPDATE-USERSTATE sp Username sp Conversation sp state
Invalid state error	ERROR sp msg_ID sp INVALID-USER-STATE sp Username sp Conversation EOM
List users logged onto server	LIST-USERS sp msg_ID EOM
List public conversations on server	LIST-CONVS sp msg_ID EOM
List users in a specific conversation	LIST-CONVUSERS sp msg_ID sp Conversation EOM
Sends an updated list of users after a user logs on or off	UPDATE-USERSLIST sp (Username)* EOM
Sends an updated list of users after a user joins or leaves a conversation	UPDATE-CONVUSERSLIST sp Conversation sp (Username)* EOM

Sends an updated list of conversations after one is created or destroyed	UPDATE-CONVSLIST sp (Conversation)* EOM
Invite a user to a one-on-one private conversation	INVITE sp Username sp Conversation EOM
Pings a user (client should not respond)	PING