Team Contract

Team Contract:
Goals:
Team Goals: To receive an acceptable grade, and to work especially hard on our GUI to make it useful and intuitive, since all of us are interested in learning more about GUI design.
Personal Goals: Each of us will work on our own to define our individual goals.

Obstacles: We expect to encounter time constraints, and the need to learn new ideas and concepts in Java.
Our team agrees that we'd like to work hard and try to get an A, and not decide that a B will be acceptable. It is acceptable, but not ideal for one or two team members to do more work than others in order for the team to get an A.

Meeting Norms:
We have no specific preference for meeting locations. We'll try to divide meetings so that some occur near each team member.
We'll try to meet briefly every day, but realistically we expect it will be closer to every other day.
We expect most meetings to be between a half hour and an hour.
Eating will be acceptable.
We plan to use Subversion to track the agenda, minutes, and action lists from each meeting.

Work Norms:
We expect this project will take about 20 hours per week per individual to complete.
We will divide different tasks in the project equally among different group members.
We'll decide on different deadlines for tasks as they fit with other tasks at our team meetings.
Individuals that are more interested in different tasks will be assigned to them as can possibly be done.
We'll record who is responsible for what in our meeting notes.
If people don't follow through we'll adjust our plan accordingly to get the work done.
At meetings, team members will describe their code, their approach, and we'll discuss the quality and effectiveness.
If we have disagreements on the quality of work on the project, we'll ask questions to TAs and try to resolve it.
If team members aren't doing their share, we'll discuss that with them individually and try to fix the problem.
We'll work at different times to fit with all of our team members individual habits.

Decision Making:
For major decisions, we should have a reasonable consensus from all team members. That means a majority of the group should support the design, and the rest of the group should also be able to work with that decision.
If one team member fixates on a particular idea, we'll present alternatives and discuss it.

Second Meeting Scheduled for:
This coming Sunday night.

Abstract Design

In our initial implementation, we expect to create a system in which users primarily conduct two-way conversations with each other. A conversation is initiated by one user attempting to send a message to another user (similar to dialing a phone number). We will also allow for chat rooms, which are publicly accessible forums with arbitrary numbers of conversation participants. Unlike two-way conversations, chat rooms can be joined by anyone who knows the name of the room.

In the long run, we hope to implement buddy lists as well, though this feature will be added after a basic system is working. Buddy lists will be maintained on the client-side, and will not persist between IM sessions. Associated with buddy lists are sign-in and sign-out messages: if user X signs in or out, all users who have user X on their buddy lists will be notified.

We also hope to implement a number of extra features. In order of priority, these are:

1. Forwarding messages to cell phones
    - In a buddy list, a user may specify a cell phone number and carrier for a particular buddy. When that buddy is offline, messages will be delivered to that number via the email associated with that cell number and carrier.
2. Typing notifications
    - A user receives notification that the user on the other end of a two-way conversation has started, stopped, or paused his/her typing of a message.
3. File-sending
4. A "Hello" protocol by which the server determines whether or not each client is still actually connected.
5. User invisibility
    - An online user can choose not to appear on others' buddy lists.

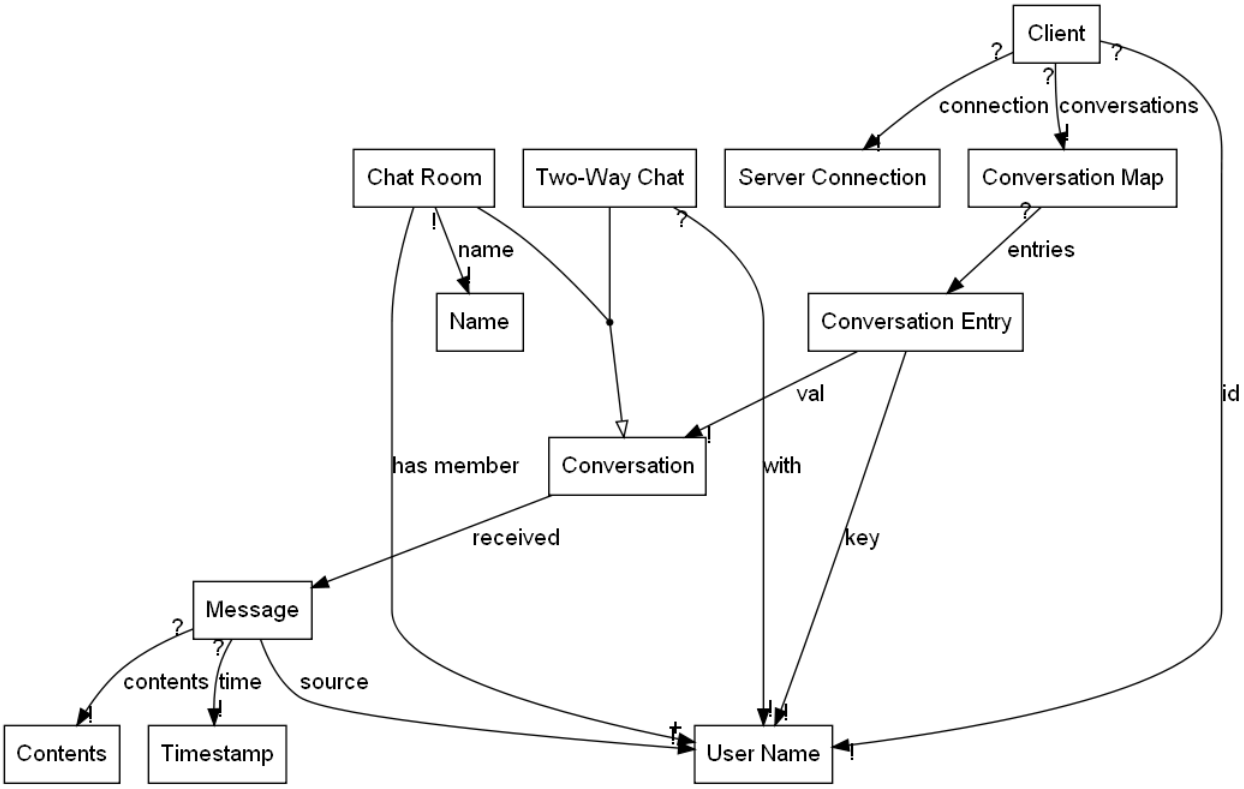
To initiate a conversation, a user "dials" another user by opening a dialog box and typing the name of the person he or she wants to send message to. Each client maintains a list of conversations, while the server simply acts as a switch that forwards received messages from a sender on one channel to a recipient on another channel. The server knows nothing of conversations; the client maintains a list of past messages from a particular user or chat room as a conversation (each conversation has its own window). When the client receives a message, it displays it in the correct conversation window based on username or chat room name. If the client receives a message from a username it hasn’t seen before, then it opens a new conversation window (prompting the user to accept if the sender is not in the user's buddy list).

We chose this model over having the server manage conversations as pairings of users and maintain one connection for each end of the conversation. For the latter model, the server would need to maintain much more state about the currently connected users. This would mean more threads, since each socket needs its own thread listening for incoming data, and concurrency would be a bigger problem to deal with. Additionally, then we would have to have the client signal the server when a conversation is over, and when that should happen is not clear. In our design, the server can function with minimal knowledge of what goes on at the endpoints, and need not worry about proliferation of running threads and open sockets.

Object models:

Client:
Descriptions:
Client - an IM client instance
Server Connection - a communication channel with the server
User Name - a string identifying some current user of the IM network
Conversation Map - a map of usernames to conversations
Conversation Entry - an entry in the conversation map
Conversation - a collection of past messages received from a particular user
Message - a record of a previously received message from another user
Contents - an actual message string
Timestamp - an indicator of message arrival time
Chat Room - a multi-way conversation
Two-Way Chat - a standard chat conversation between two people which cannot be joined by anyone else
Name - the publicly accessible name of a chat room (and the name by which it can be joined)

Relationships:
connection - the server connection for this client
conversations - the set of currently active conversations for this client
entries - the set of key-value mappings in this map
key - the key for a map entry
val - the value for a map entry
with - the username on the other end of a two-way chat
name - the name of this chat room
has member - one of the users currently in the chat room
received - a message that was received as part of this conversation
id - the username the client chose at logon
contents - the contents of a logged message
time - the timestamp of a logged message
source - the sender of a logged message


Client
connection
conversations
Chat Room
Two-Way Chat
Server Connection
Conversation Map
name
entries
Name
Conversation Entry
val
id
has member
Conversation
with
received
key
Message
contents
time
source
Contents
Timestamp
User Name

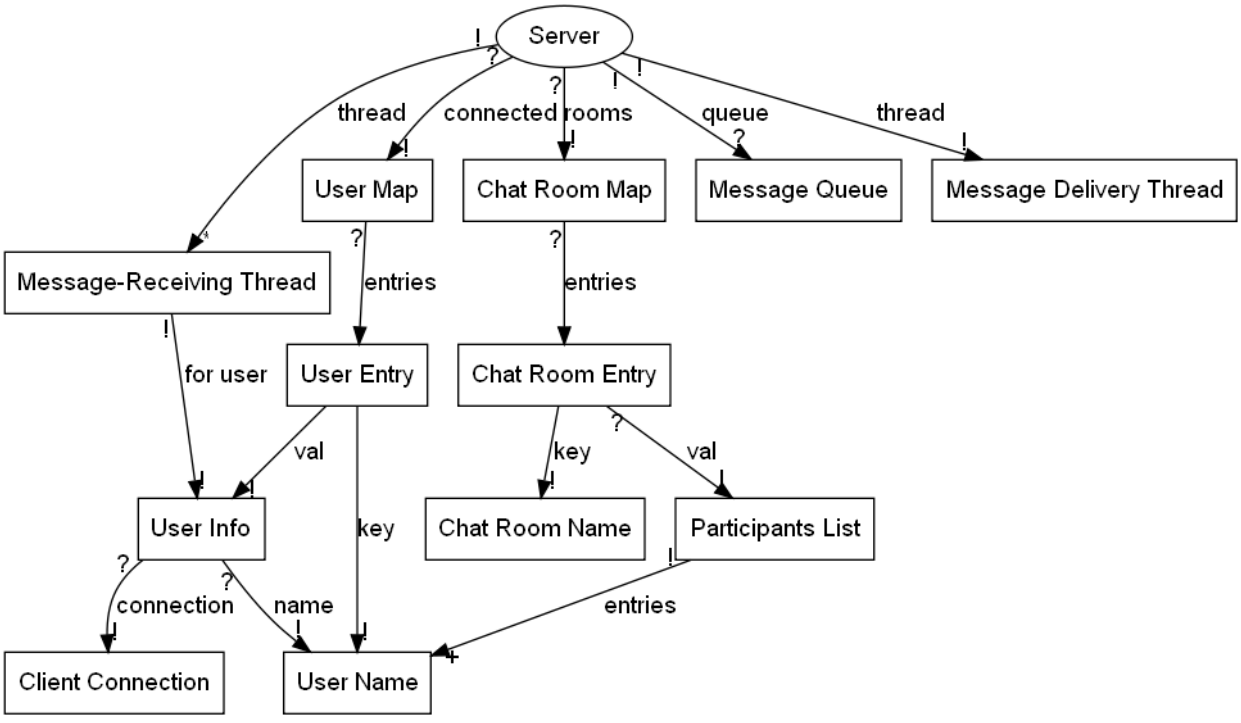
Server:

Descriptions:

Server - an IM server instance
User Map - a map of usernames to user info
Chat Room Map - a map of chat room names to lists of chat participants
Message-Receiving Thread - a thread to receive messages from a particular user
Message Delivery Thread - the one thread for this server instance that processes messages awaiting routing, and requests awaiting treatment
Message Queue - a FIFO queue of undelivered messages and unprocessed requests
User Entry - an entry in the user map
User Info - a data structure containing information about a particular user
User Name - a string identifying some current user of the IM network
Client Connection - a communication channel with a particular client
Chat Room Name - a string uniquely identifying a particular chat room
Chat Room Entry - an entry in the chat room map
Participants List - a list of current chat room participants for a particular chat room

Relationships:

thread - a thread running on the server
connected - the users who are currently connected
queue - the queue of pending messages/requests
entries - the entries of this map or list
key - the key of a map entry
val - the value of a map entry
for user - the user from whom this thread receives messages
connection - the client connection associated with a particular user
name - the name associated with a particular user



Client/Server Protocol

All communications between client and server will take place via exchanges of XML messages.

There are two types of XML snippets that a client may send to the server:

- A *message* consists of content to be sent to another user. Every message XML snippet contains a destination indicating what user it is to be sent to. There may be different types of messages - contentful messages, messages about typing status, chat room messages, and so on.
- A *request* is a command sent to the server, such as a login request or a request to join a chat room. The contents of this request XML document depend on the type of request.

The server also sends two types of XML snippets:

- A *message* is as defined above. When a server forwards a message to the appropriate destination(s), it inserts a tag indicating the user who was the source of this message.
- An *acknowledgment* (or ACK) is as it sounds - a simple indication to a client that an XML document sent by the client was received. Every ACK has a status element, which is "OK" if the operation requested by the client succeeded, and otherwise contains a string indicating what error occurred.

When a client sends a message for which it expects an ACK, it hangs momentarily waiting for the response, and times out if it does not receive one quickly. (An error message is displayed on timeout.) No acknowledgment is given for requests to log out or to leave a chat room.

There is no request type for creating a new chat room; when someone requests to join a room by a particular name, the room is created on the server if it does not exist already. The room is destroyed when the last person leaves it.

Below are some sample client-generated XML messages.

```
<LoginRequest>
  <username>Group03</username>
</LoginRequest>

<LogoutRequest>
  <source>Group03</source>
</LogoutRequest>

<Message type="2way">
  <dest>Group04</dest>
  <contents>This is a message.</contents>
</Message>

<Message type="chat">
  <dest>chatroom2008</dest>
  <contents>This is a message.</contents>
</Message>
```

```
<Message type="starttyping">
  <dest>Group04</dest>
</Message>

<Message type="pausetyping">
  <dest>Group04</dest>
</Message>

<Message type="endtyping">
  <dest>Group04</dest>
</Message>

<JoinChatRoomRequest>
  <room>chatroom2008</room>
</JoinChatRoomRequest>

<LeaveChatRoomRequest>
  <room>chatroom2008</room>
</LeaveChatRoomRequest>
```

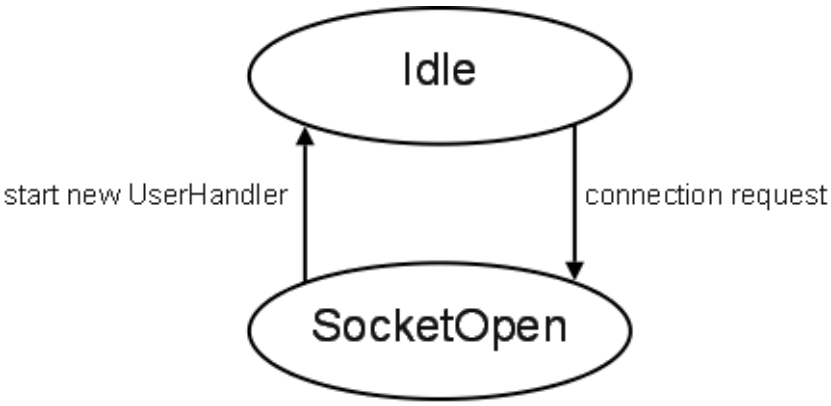
Some sample server-generated XML messages.

```
<Message type="2way">
  <source>Group03</source>
  <contents>This is a message.</contents>
</Message>

<Message type="chat">
  <source>Group03</source>
  <dest>chatroom2008</dest>
  <contents>This is a message.</contents>
</Message>

<LoginAck>
  <status>OK</status>
</LoginAck>

<LoginAck>
  <status>TAKEN</status>
</LoginAck>

<MessageAck>
  <status>OK</status>
</MessageAck>

<MessageAck>
  <status>OFFLINE</status>
```

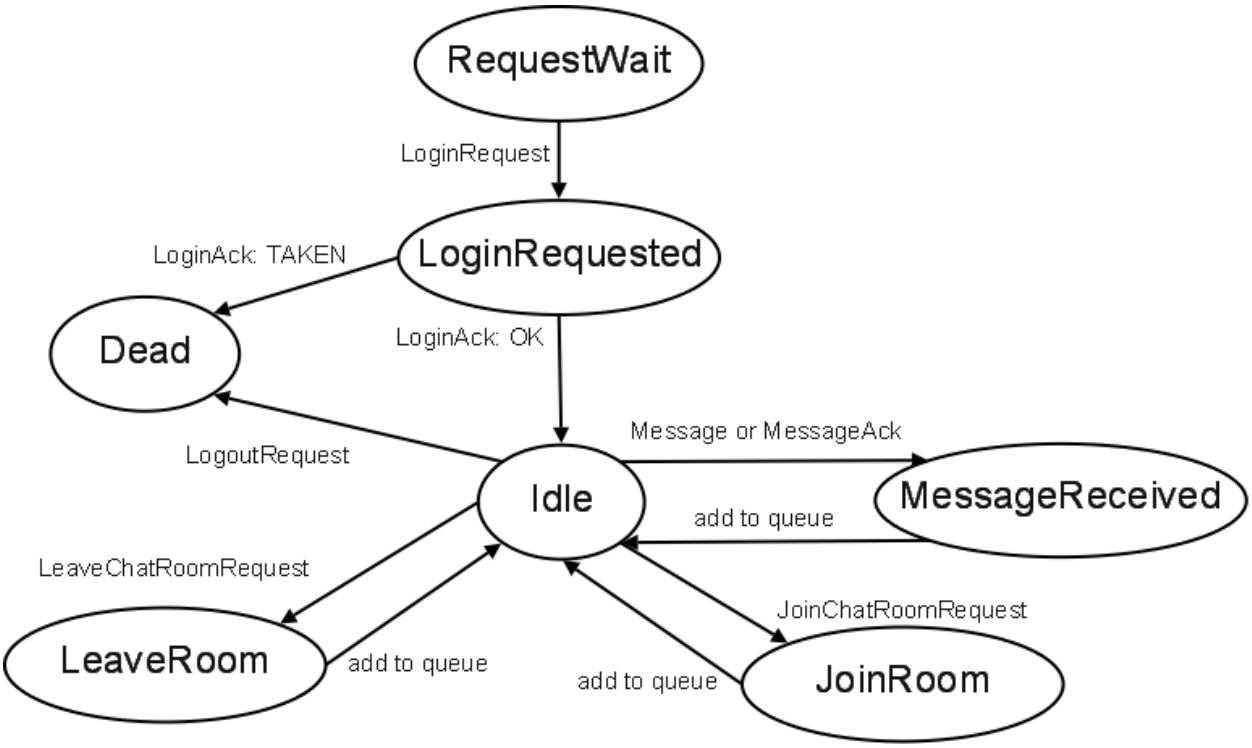
```
</MessageAck>

<MessageAck>
  <status>MALFORMATTED</status>
</MessageAck>

<JoinChatAck>
  <status>OK</status>
</JoinChatAck>

<JoinChatAck>
  <status>CREATEFAILED</status>
</JoinChatAck>
```

State Machines:

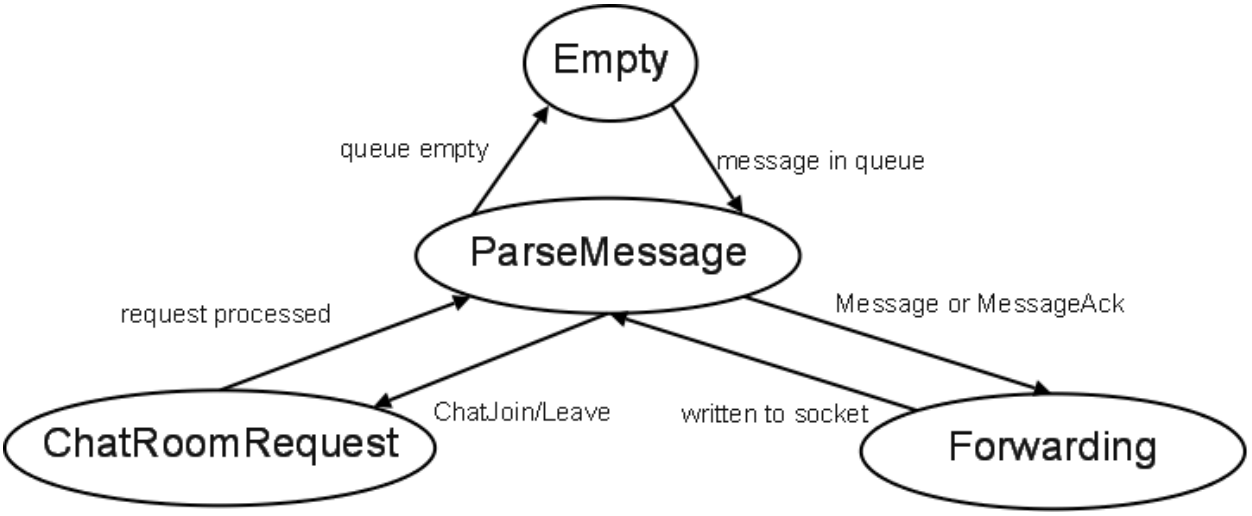
In addition to our XML text based protocol, we've created a number of state machines to help represent the client and server in our models. First, we have the state machine for the main server thread. The server starts in an idle state, awaiting connections on a given port. When a connection is received, the socket is opened, and a new UserHandler thread will be created to initialize and receive messages from this user.



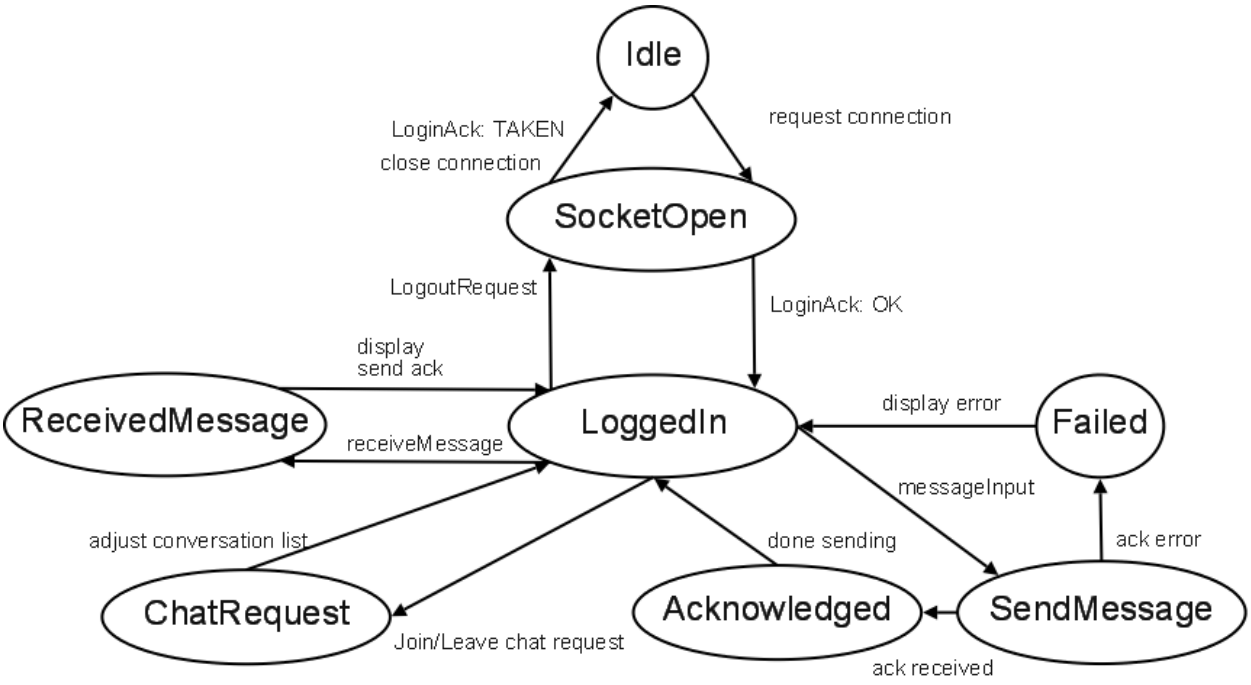
Next, we created a state machine for the UserHandler thread. To start, this thread will be waiting for a login request. When this is received, if the name is taken, the connection will be determined and the thread will end. If it is not taken, this user will be added to the server, and the thread will enter an idle state, waiting to receive messages. Messages to be forwarded to other users will be added to the queue, messages to join or leave a chat room will be processed properly, and a LogoutRequest will close connections and terminate the thread.



Next, we have a state machine for our message queue. This will run in a separate thread. When this thread is notified that there is a message in the queue, it will parse the message. If it needs to be forwarded, it will write to the appropriate socket, and if it is a chat room request, it will process that locally by adjusting the associated data structures.



Finally, we need a state machine for the client. The client begins in an idle state, and then attempts a login request. If acknowledged OK, the user is logged in, and if not, they are disconnected and can try again. Once logged in, they can receive messages, which need to be properly displayed and replied to with an acknowledgement, make chat requests to the server, which will change their local list of conversations, as well as send messages, which should be acknowledged by the users that they are intended for, with an error otherwise appearing.


Idle
request connection
LoginAck: TAKEN
close connection
SocketOpen
LogoutRequest
LoginAck: OK
display
send ack
ReceivedMessage
LoggedIn
display error
Failed
receiveMessage
messageInput
adjust conversation list
done sending
ack error
ChatRequest
Acknowledged
SendMessage
Join/Leave chat request
ack received

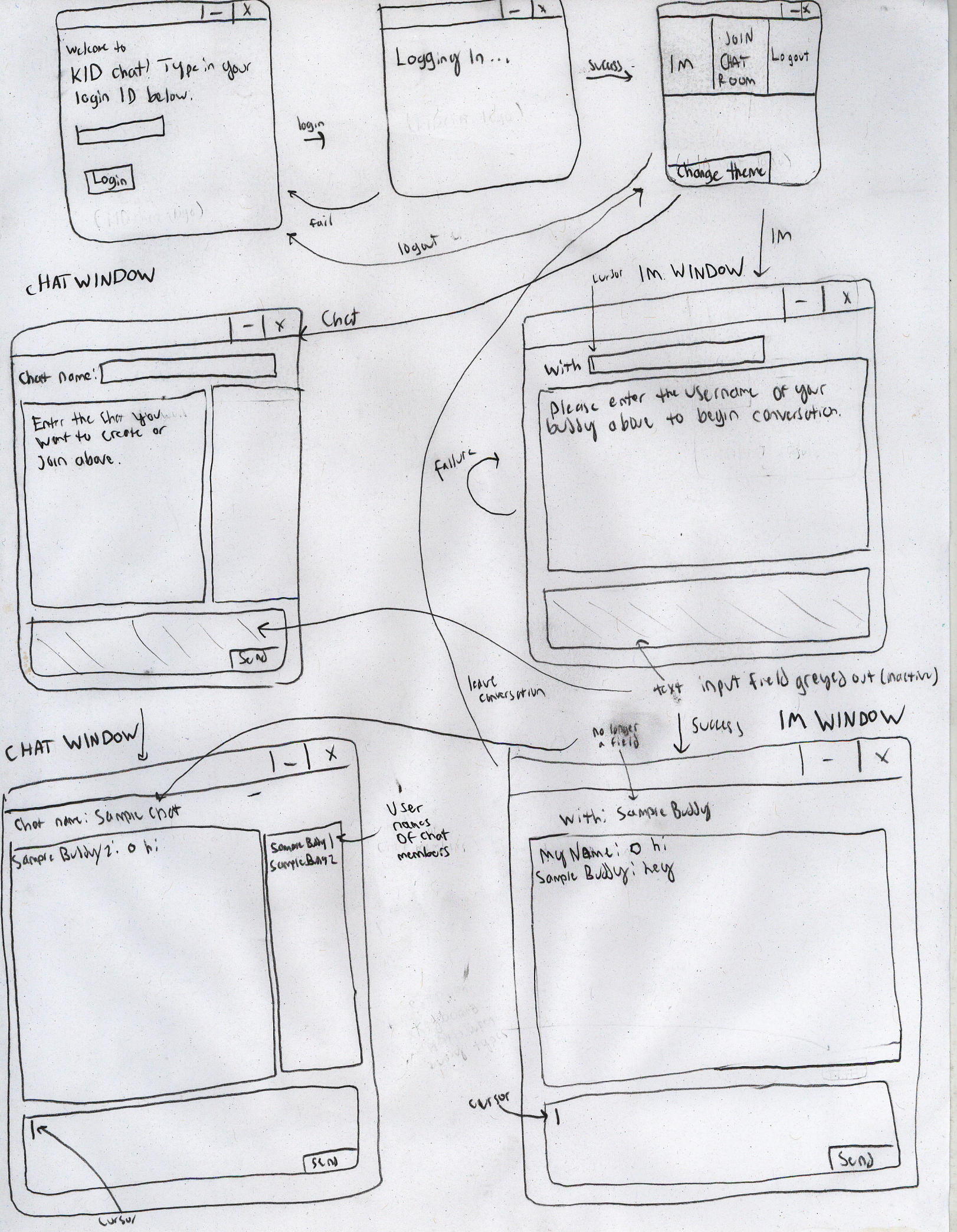
Usability Design

The GUI for our basic implementation is shown below. A basic state-model type diagram shows the user operations that produce each box to display. In general, upon opening the program, a log in box is displayed with a prompt for username. While the logging in process is occurring, the user is presented with a “logging in” box that disappears after successful login and is replaced with the main program box. If login fails or the user presses the “logout” button, the user is returned to the login box.

If the user presses the IM button, a new IM window appears where the user can enter in the user name of the user he/she wants to IM. At this point, the box where the user can type messages to IM is disabled. Upon failure (user is not logged in), the box closes and re-opens. Upon success, the field where the user can type messages becomes enabled and the field where the user had typed the username becomes no longer modifiable. The user can type messages, which appear in the history box above after successful sending, along with the user’s friend’s reply.

If the user presses the Join Chat button, a new Chat Window appears where the user can enter in the name of the chat he/she would like to join or create. Upon entering in a name, the Chat Window will begin to display the users in the particular chat on a pane on the right (including the user that initiated the chat), make the chat name not modifiable, place the cursor in the new message box, and begin displaying messages from the users friend(s) in the history box.

Advantages of this design model include only a few different boxes needed to be created (A login/main box, a chat box, and an IM box) and simplicity that creates a relatively intuitive user interface.


Welcome to KID chat! Type in your login ID below.
Login
login
Logging in...
success
fail
logout
IM
JOIN CHAT ROOM
Logout
Change theme
IM
CHAT WINDOW
Chat
Chat name:
Enter the chat you want to create or join above.
Send
Cursor
IM WINDOW
With
Please enter the username of your buddy above to begin conversation.
failure
leave conversation
text input field greyed out (inactive)
success
CHAT WINDOW
Chat name: Sample Chat
Sample Buddy2: o hi
SampleBuddy
SampleBuddy2
User names of chat members
Send
cursor
IM WINDOW
no longer a field
With: Sample Buddy
My Name: o hi
Sample Buddy: hey
cursor
Send

The GUI for extensions on our implementation is shown below. Upon logging in, the user is presented with a buddy list (top left) instead of our simpler buddy menu (previous page, top right). This buddy list will be blank upon login (will not be stored between sessions, although this is also an extension we could possibly create), and the user can press the "Add Buddy" button, upon which he/she will be given a popup prompt to add a buddy username and possibly a mobile phone number and carrier for use in SMS offline messaging. This will result in the buddy's username being shown on the buddy list in the "online", provided the buddy is logged in and has set himself/herself as visible, and in the buddy list's "offline" section otherwise.

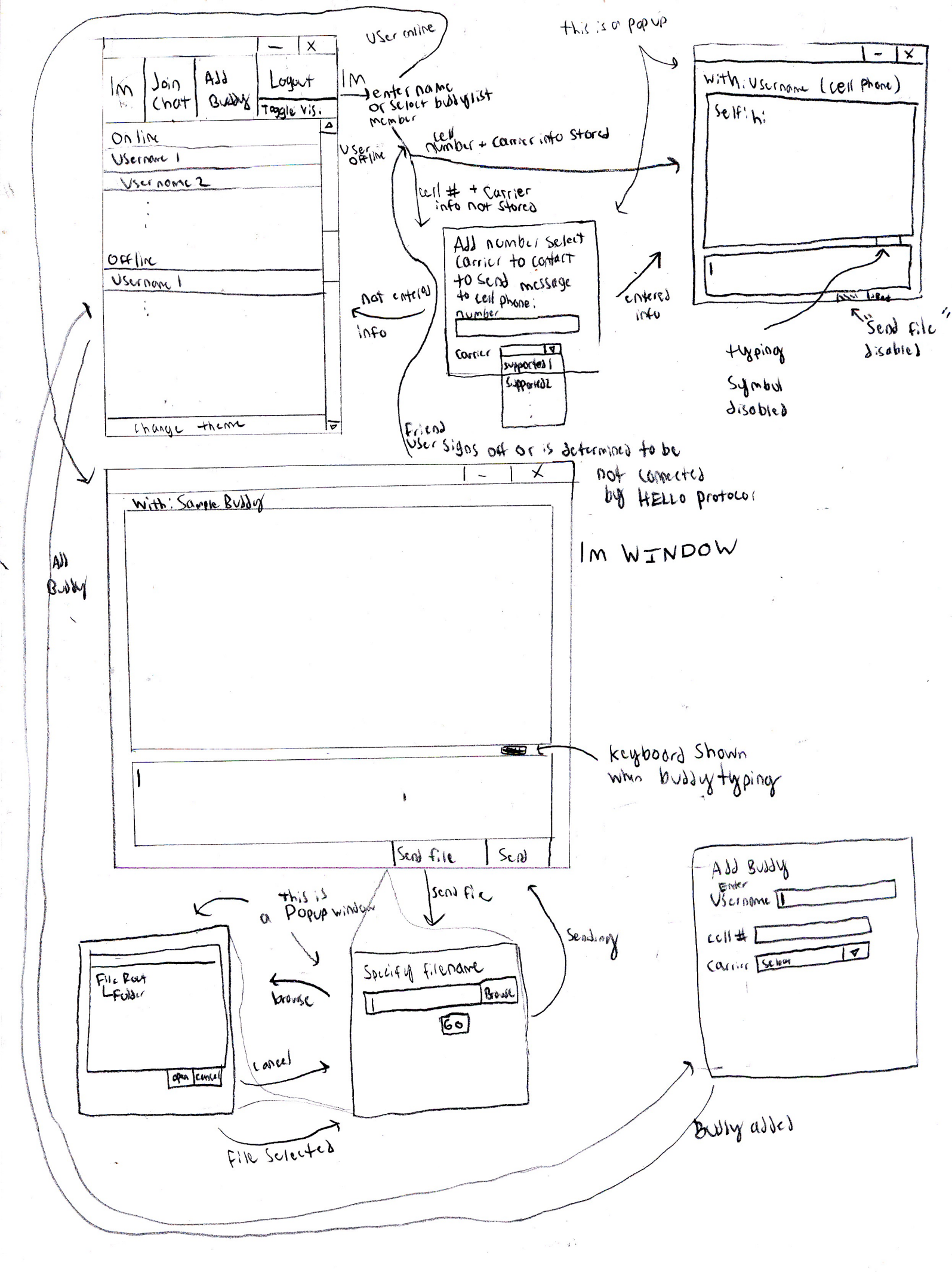
Visibility is a possible extension of a buddy list; a user can toggle visibility by pressing the "Toggle Visibility" button on the buddy list box; if visibility is off, this user will not appear on other user's buddy lists as being online.

Upon pressing the IM button and entering in the username of a buddy in the IM window or selecting a buddy from the buddy list, three possible cases can occur:

1) The buddy is online; the IM window then updates as normal (shown in the center of the page).
2) The buddy is offline, and a cell phone number and carrier have been stored for this buddy; the IM window then updates to a reduced-functionality IM window where the user can only send messages (as opposed to files) and cannot see whether the buddy is typing (since our implementation will not allow replies from cell phones). More details sending files and typing notifications will be discussed below.
3) The buddy is offline, and a cell phone number and carrier have not been stored for this buddy; a popup window asking for the addition of this info is displayed, and upon successful entrance of information case a reduced functionality window is provided as discussed in case (2).

Our extensions in a normal IM Window consist of

1) Buddy typing notification; a keyboard is shown directly above the message entrance box in the IM window while the buddy is typing.
2) Send file; there is a "send file" button next to the regular "send" button used to send a file; upon pressing this button, a popup window prompts for specification of filename, which has a "browse" button allowing the browsing of the user's local file system to select a file if the user does not know the exact path. Upon successful selection of a file, the filename field is then auto-completed using the specified file; upon filling the filename file and pressing the "send" button, the "send file" button is replaced by a "sending file" string specifying that the file is being sent (that returns to being a normal "send file" button after the file is completely sent). While specifying a filename, canceling brings the user back to the IM window.


IM
Join Chat
Add Buddy
Logout
Toggle vis.
Online
Username 1
Username 2
Offline
Username 1
change theme
User online
IM
enter name or select buddylist member
User offline
cell number + carrier info stored
cell # + carrier info not stored
this is a popup
With: Username (cell phone)
Self: hi
typing
Send file disabled
Symbol disabled
Add number/select carrier to contact to send message to cell phone:
number
Carrier
supported 1
supported 2
not entered info
entered info
Friend User signs off or is determined to be not connected by HELLO protocol
With: Sample Buddy
IM WINDOW
All Buddy
keyboard shown when buddy typing
Send file
Send
this is a Popup window
send file
Sending
File Root Folder
open
cancel
browse
Specify filename
Browse
GO
cancel
file selected
Add Buddy
Enter Username
cell #
carrier
Select
Buddy added