Lab 4

Chatting AIMlessly

IM Client Implementation Project --- Part 1
Due February 28/March 1, 11PM

This week, you will begin work on a project we will complete next week, the construction of your own IM chat client. The program you develop this week will have several limitations. For example, all chat messages you send to or receive from others will be displayed in a single window. Next week, we will address these limitations.

This week's lab will have two distinct parts. First we will ask you to complete a tutorial that introduces more of the features BlueJ provides to help you debug programs. Then, we will ask you to work on the development of an IM client. The debugging tutorial is not included in this handout. We will distribute a separate handout for the tutorial in lab on Monday and Tuesday.

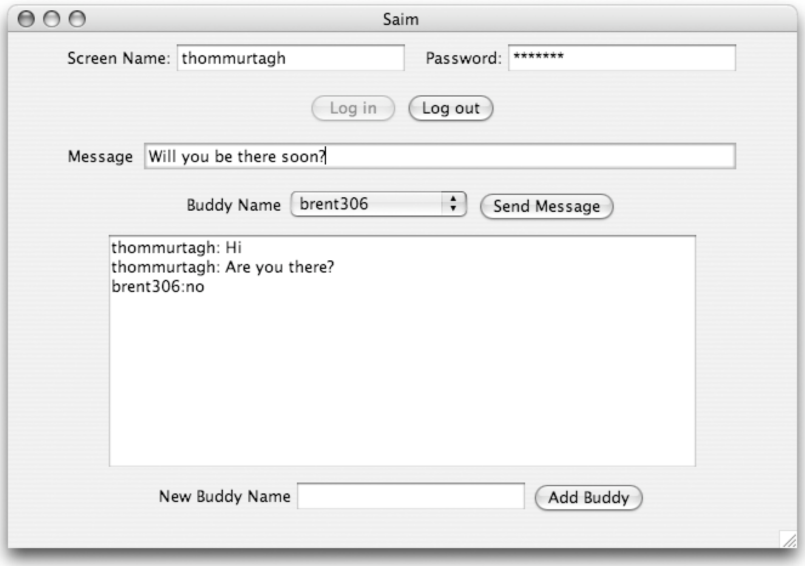
Pre-lab Preparation

You should always prepare for lab by carefully reading the lab assignment and sketching out both the code you will write and your plan for implementing and testing various components of the complete program. This week, there is also something a bit more mundane you should do as part of your preparation.

As you develop your IM client, you will need to periodically test the code you have written by exchanging messages with another IM client. Once your programs are working fairly well, it will be possible (and maybe even enjoyable) to do this by exchanging messages with other students in the lab or with friends who are nowhere near our lab. Early on in the process, however, it will be much simpler if you can exchange messages with yourself by running a standard IM client on your machine at the same time you are testing your own IM program. All the Macs in our lab have Apple's IM client, iChat, and the standard AIM AOL client installed. So, you will be able to start up AIM or iChat, then start your program and send messages back and forth from one side of your computer screen to the other between the two programs.

To do this, you will have to log in with different screen names on your IM program and on the other IM client you use. Accordingly, if you don't already have more than one IM screen name, we would suggest you register for a two additional IM accounts before coming to lab. For one of these accounts, choose a screen name that does not contain any blanks or capital letters. For the other, choose a screen name that contains several blanks and capital letters. You will use this second screen name to test one feature included at the end of the lab.

In addition, use a standard IM client to log into each of your accounts and make sure to add the other accounts as buddies. Also, add a few of your usual buddies to the new accounts so that they have at least three or four names in their buddy lists.



Screen Name: thommurtagh Password: *******

Message: Will you be there soon?

Buddy Name: brent306

```
thommurtagh: Hi
thommurtagh: Are you there?
brent306:no
```

New Buddy Name

Client Interface

A sample of what the program's interface might look like is shown on the right.

Across the top of the program's window, there are fields used to enter the user's IM screen name and password. Below these fields are buttons used to log in and out. Obviously, the user is supposed to enter a screen name and password before pressing the log in button. Also, until log in is completed successfully, the buttons for logging out and sending messages are disabled. They should become enabled once the user logs in and then be disabled again upon logout.

Next, there is a text field intended to let the user enter a message to be sent to one of his or her buddies. Below this there is a pop-up menu labeled "Buddy Name". The menu is intended to include the names of the user's buddies. This menu will initially be empty. The processes used to add buddy names to the menu are described below.

Once a message has been entered and a buddy selected from the buddy menu, the user can send the message by clicking on the "Send Message" button. In addition, to provide a more convenient interface, your program should also send a message if the user presses the return key while typing in the message field.

Immediately below these items, there is a text area that occupies about half the program window. This area is used to display the messages sent between the program's user and his or her buddies. When the program's user sends a message, the program should display the user's screen name together with the message in this large text area. Similarly, when a message is received from a buddy, the buddy's name and the message should be displayed.

When your program is complete, it will automatically fill the menu before the "Send Message" button with the names of all of your buddies who are currently online. To make your program usable before this feature is implemented, we also want you to include a mechanism that will provide a manual way to add names to your buddy menu. At the bottom of your program's window, there should be a text field labeled "New Buddy Name" and a button labeled "Add Buddy". Whenever the button is pressed, your program should add the contents of the text field to the buddy menu.

OSCAR is a Grouch!

In order to write an IM client, you have to learn a little about the protocol IM clients use to communicate. There are actually several protocols involved. The standard AIM client and other clients that AOL blesses (like Apple's iChat), use a protocol named OSCAR (Open System for CommunicAtion in Realtime). Unfortunately, OSCAR's name is inaccurate. There is nothing "open" about it! OSCAR is a proprietary protocol that AOL guards jealously. Those who write IM clients using OSCAR have to modify them regularly to cope with changes AOL introduces mainly so that non-standard clients stop working. After some complaining, AOL introduced a non-proprietary protocol that can be used to write IM clients. This protocol is named TOC (Talk to OSCAR?). You will be using TOC.

TOC is a text-based protocol. All it requires is sending and processing text commands similar to those used in protocols like POP and SMTP. Much like POP and SMTP packets, each packet you send to a TOC server will begin with a code that indicates the type of the message being sent. We will begin by giving an overview of the types of packets your program will exchange with the TOC server. Then, we will discuss the details of the format of the contents of these packets.

Your program's interaction with AOL's TOC server will begin with the transmission of a "toc2_signon" packet that contains both the user's screen name and password. If AOL accepts this packet, the TOC server will respond by sending a packet that starts with the prefix "SIGNON". If the screen name or password are invalid, the TOC server will respond with a packet that starts with "ERROR". One additional packet is required to complete the login process. If your program receives a "SIGNON" message from the server it must send a "toc_init_done" message in response.

Once your program has completed the login process, it can transmit messages to the user's buddies by sending "toc2_send_im" packets. Each such packet will contain the body of a message and the screen name of the intended recipient. Unlike POP and SMTP servers, a TOC server will not respond to each such message with a packet indicating success or failure. If the message is rejected, the server may send an ERROR packet, but if it is accepted the server will not send any response.

When another user sends an IM message to the person using your program, the TOC server will send your program an "IM_IN2" packet containing the message and the sender's screen name.

While you are connected, the TOC server will also send you a variety of other messages. In particular, whenever someone included in the user's buddy list goes offline, come online, puts up an away message, etc. your program will receive an "UPDATE_BUDDY2" message.

Finally, there is no special “QUIT” message that must be sent to terminate the connection. Instead, when the user of your program presses the logout button your program will simply close its connection to the server.

TOC to me

As in POP and SMTP, the packets sent using the TOC protocol consist of a packet type name followed by one or more arguments. For example, the packet type used to send an IM message is “`toc2_send_im`”. The arguments for such a packet include the recipient's screen name and the message. Thus, to send Tom the message “Hi there”, you might send the packet

```
toc2_send_im thommurtagh "Hi there"
```

In packets sent by the client to the server, the arguments are separated from the packet type and from one another by spaces. Arguments (such as the actual IM message) that contain special characters (including spaces) must be quoted.

Screen names in packets sent to the server should be “normalized”, meaning that all letters must be lower case and all blanks eliminated. Initially, you should be able to meet this requirement without writing much code. First, make sure that you don’t try to talk to anyone whose screen name contains blanks. In addition, use the `toLowerCase` method to eliminate any upper case letters in a screen name before you place it in any message you send to the server. Near the end of this lab, you will add instructions to complete the normalization process by removing all blanks from the screen names you send.

Receiving IM Messages

When someone sends you a message, the server will send your client program a packet that starts with the prefix “`IM_IN2:`”. This prefix will be followed by the sender's screen name, several short arguments, and the actual contents of the sender's message. A sample of such a packet is shown below:

```
IM_IN2:ThomMurtagh:F:F:still there?
```

In packets sent by the server to the client, the arguments are separated from one another using colons and screen names are not normalized. Arguments (such as the actual IM message) that might contain special characters (including colons) are always positioned last in the list of arguments so that no quoting is needed.

The argument after the screen name in an `IM_IN2` packet indicates whether or not the message is an away message. `T` indicates the message is someone's away message. `F` indicates the message was sent to you by an active user. When your program receives such a packet, it should display the screen name for the sender and the body of the message in the text area included in the program’s window.

There is one little complication that arises when you are communicating with a person using one of the standard IM clients (AIM or iChat). These program encode the text of the messages users send using HyperText Markup Language (HTML), the language used to describe web pages. This makes it possible to send messages using different fonts or to display the text of messages in variety of colors, if the receiving client knows how to interpret the HTML. Your client will not know how to do this.

With HTML included, the actual `IM_IN2` packet your client receives when a users sends the “still there?” message used as an example above might look like:

```
IM_IN2:ThomMurtagh:F:F:<HTML><BODY BGCOLOR="#FFFFFF"><FONT ABSZ=12 SIZE=3>still there?</FONT></BODY></HTML>
```

If you examine the contents of this packet for a moment, you will notice that it contains many less than and greater than signs. HTML uses a less than sign to indicate the beginning of each of its commands and a greater than sign to indicate the end. As a result, even if you don’t understand HTML, it is easy to remove all the HTML from an incoming `IM_IN2` packet. You simply have to remove all of the text found between less than signs and greater than signs together with the less than and greater than signs themselves. In the example shown above, all the HTML tags appear before or after the text of the actual message. **This will not always be the case.** Your code will need to remove HTML tags even if they appear in the middle of the text of a message.

Good Buddies

There are a collection of packet types used to manage buddy lists (including things like blocking other users) which we will not consider in this project. There is one packet that provides buddy information that we will find very useful. Whenever one of your buddies changes state (i.e. logs on, logs off, puts up an away message, etc.) the server will send your client an "`UPDATE_BUDDY2`" packet. A sample of the contents of such a packet is shown below:

```
UPDATE_BUDDY2:WSJ:T:0:1127376565:0: O:0
```

Each such packet will have seven arguments. The first argument in each `UPDATE_BUDDY2` packet will be the screen name of the buddy whose status is being reported. The screen name will not be normalized since this packet is coming from the server. This screen name will be followed by a colon and either a `T` or an `F`. A `T` indicates that the individual with the screen name is now online. An `F` indicates the individual is not online. If the individual is online, the remaining fields provide details about whether he or she idle or displaying an away message. We will not attempt to describe these other fields in greater detail.

Using these packets, you can automatically add and remove names from your program's buddy menu so that it will contain the screen names of all of your buddies who are actually online. To do this, whenever an "`UPDATE_BUDDY2`" packet with a `T` after the buddy's name arrives you should add the name to the menu (if it isn't already there), and whenever an update with an `F` arrives, you should remove the name from the menu.

The tricky part of this is that AOL sends you so many buddy update packets that you are likely to end up with many copies of a buddy's screen name in your menu if you are not careful. You could avoid this by checking to see if a name is in the menu before you add it. A simpler approach, however, is to simply remove a name before you try to add it. It may seem odd to remove a name without first checking that it is actually in the menu, but the `removeItem` method doesn't complain if you try to remove an item that isn't actually there.

Make No Mistake

One type of packet the server may (regrettably) send to your client is the "`ERROR`" packet. After the packet type "`ERROR`", you will find a colon and one of the following error codes.

General Errors

901	- <screenname> not currently available
902	- Warning of <screenname> not currently available
903	- A message has been dropped, you are exceeding the server speed limit

Administrative Errors

911	- Error validating input
912	- Invalid account
913	- Error encountered while processing request
914	- Service unavailable

IM & Info Errors

960	- You are sending message too fast to <screenname>
961	- You missed an im from <screenname> because it was too big.
962	- You missed an im from <screenname> because it was sent too fast.

Authorization errors

980	- Incorrect nickname or password.
981	- The service is temporarily unavailable.
982	- Your warning level is currently too high to sign on.
983	- You have been connecting and disconnecting too frequently. Wait 10 minutes and try again. If you continue to try, you will need to wait even longer.
989	- An unknown signon error has occurred

Your program will only need to react to `ERROR` packets received in response to a signon packet. Your program does not need to take any specific action to react to other `ERROR` packets, but you should probably display all error packets received to help with debugging. In particular, note that if error code 983 appears, trying to fix your code and running it again will only make things worse. Instead, you should either take a break or switch to using a different IM account.

TOC Sign On packets

When a user first connects to the TOC server, your program will have to send a TOC signon packet. An example of what a TOC signon packet looks like is shown below:

```
toc2_signon login.oscar.aol.com 5190 murt74 0x3a060d5c english "TIC:SAIM" 160 98200960
```

Obviously, these packets are quite ugly! In fact, they are so ugly that we will provide a special class to help you construct them. This class is described below. It will make it unnecessary for you to actually understand the contents of a TOC signon packet. For completeness, however, we will still provide a brief description of the fields of a signon packet.

As the example above suggests, each signon packet begins with the packet type "`toc2_signon`". This is followed by the name and port number for the OSCAR server that the TOC server will use to actual send and receive your message. Next, the packet contains the user's screenname and password (in a weakly encrypted form). Finally, there are four fields of dubious value. The first specifies the language to be used, but the only supported option is "english". Next comes a quoted string identifying the name of the client program. Then come two mysterious numbers. The first is always 160 and not explained by the TOC documentation. The second is derived from the first letters of the user's screenname and password by following a peculiar little algorithm.

To make it easy to send TOC login packets, we will provide you with a class named `TOCSignonPacket`. This class is quite simple to use. To create a new `TOCSignonPacket`, you evaluate a construction of the form

```
      new TOCSignonPacket( screenname, password )
```

The screen name provide should be normalized.

There is only one method associated with this class, `toString`. Invoking this method produces the text of a valid `toc2_signon` packet for the screen name and password provided in the construction. Therefore, to create a signon packet, you can use instructions of the form:

```
      TOCSignonPacket signon = new TOCSignonPacket( screenname, password);
      String packetContents = signon.toString();
```

You could then send the string `packetContents` to the TOC server.

While the `TOCSignonPacket` class makes it unnecessary for you to understand the format of a login packet for this assignment, you will need to know how the server will respond when you send it such a packet. If there is something wrong with the packet, the server will send you one of the "`ERROR`" packets described above. In this case, you should simply inform the user that the login failed and close your connection to the server.

If the information in the login packet is correct, the server will respond by sending a packet that begins with the text "`SIGN_ON`". As soon as your program receives such a packet, it should complete the login process by sending the server a packet of the form:

```
      toc_init_done
```

Other Server Packets

There are several other types of packets the server may send to you including a `CONFIG` packet that describes all your buddies and buddy groups. You will not need to interpret or process such packets.

The FLAPConnection Class

Unlike POP and SMTP, TOC is not based directly on TCP. Instead, TOC packets are sent using an AOL-specific protocol named FLAP (no one seems to know what FLAP stands for). The `NETConnection` class we have used in the last two labs is based on TCP. Therefore, to enable you to work with FLAP, we will provide a new class named `FLAPConnection` that will enable you to send and receive TOC packets without worrying about the details of the FLAP protocol.

To use this class and the `TOCSignonPacket` class described above you must include the line

```
import TOCtools.*;
```

at the beginning of your program.

Constructing a `FLAPConnection`

When you construct a `FLAPConnection`, you don't have to provide a server name or a port number because you will always be connecting to the same port on the same AOL server. Also, since constructing a `FLAPConnection` does not actually complete the process of logging in to AOL, no password is needed. You only have to provide the user's screenname in normalized form. The form of the construction is therefore:

```
new FLAPConnection( screenName )
```

FLAPConnection methods

Most of the methods provided by a `FLAPConnection` are quite similar to the methods provided by the `NetConnection` class. Assuming that `toServer` is a variable declared as

```
FLAPConnection toServer;
```

and associated with a `FLAPConnection` using an assignment statement, you can use method invocations of the following forms to communicate with the server.

- An invocation of the form

```
toServer.out.printPacket( someString );
```

can be used to send a TOC packet to the AOL server.

- An invocation of the form

```
toServer.in.nextPacket()
```

can be used to retrieve the next TOC packet received from the server.

- An invocation of the form

```
toServer.addMessageListener( this );
```

can be used to request that the `FLAPConnection` notify your program whenever packets arrive or the server closes the connection. Once such a request has been made, the `FLAPConnection` will execute any code you place in a method defined with a heading of the form

```
public void dataAvailable() {
```

whenever a packet is received from the server. In addition, if the server closes the connection, the `FLAPConnection` will execute any code you place in a method with a heading of the form

```
public void connectionClosed() {
```

- An invocation of the form

```
toServer.close();
```

can be used to terminate the connection. You should do this when the user presses the logout button.

Implementation Plan

You should plan to add code to implement one feature at a time and to test each feature before moving on to the next step. The following gives an possible plan for such an approach.

1. As usual, a good place to start is to simply write the code for a constructor that will build the desired interface. Use `JPanels` to keep things like the log in and log out buttons together. Test your code periodically as you add components. Make sure the correct buttons are initially enabled and disabled.
2. Next, write the code that will add an entry to the buddy menu when the user clicks "Add Buddy". Make sure that this happens only when the "Add Buddy" button is clicked, not also when the log in button is clicked.
3. Try to get logging in to work. To do this, you will need to construct a `FLAPConnection` and a `TOC-SignonPacket`. Then, you will send the signon packet through the connection. Finally, you will get a packet from the server using `in.nextPacket` and test to see if it is a `SIGNON` packet or an `ERROR` packet. If a `SIGNON` packet is received, you should disable the login button and enable the button used to send messages. Don't bother to write code to normalize the user's screen name at this point. Just make sure that you type in your screen name in normalized form when you test your program.
4. Add code to close the connection when the user presses logout.
5. Now, write a very simple version of `dataAvailable` that will just append any packet received from the server to the contents of the program's text area. Add "this" as a message listener to the `FLAPConnection` after the user logs in correctly. Then, send yourself a message from a standard IM client running on your machine to see what arrives. At this point, we haven't asked you to write any code to remove HTML tags from messages, so the messages you receive will look a bit different from what you send.
6. Modify `dataAvailable` by adding code to handle `UPDATE_BUDDY2` messages by adding or removing buddy names from the buddy menu when these packets arrive.
7. Add the code needed to actually send an IM message when the user presses "Send Message". Remember that these messages should also be displayed in your program's text area.
8. Modify `dataAvailable` so that it only displays incoming IM messages and `ERROR` packets (i.e. not `UP-DATE_BUDDY2` or other status packets).
9. Modify the code that sends `toc2_send_im` packets so that it correctly normalizes all screen names that it sends to the server.

   In our description of the TOC protocol, we mentioned that any screen names your program sends to the TOC server are supposed to be "normalized" by having all upper case characters replaced by lower case characters and all blanks removed. You should have already written some code to address this issue. The code you have written to send IM messages should use the `toLowerCase` method to remove capital letters from the destination screen names placed in outgoing `toc2_send_im` messages. Now, you need to add statements to remove all blanks from the such screen names. This will require writing a while loop. Each time the loop is executed it should remove one blank from an un-normalized screen name. The loop should execute until no blanks remain.
10. Modify `dataAvailable` so that it only shows the buddy's screen name and the actual contents of the messages received when an `IM_IN2` message arrives. This will involve writing another loop to remove the HTML from incoming messages.
11. Define a `textEntered` method that simulates the effects of clicking the "Send Message" button. Do this by invoking the `doClick` method on the send button rather than by cutting and pasting code from `button-Clicked`.

Getting Started

As usual, you should start your work by launching BlueJ. BlueJ will open last week's project by default. You should create a new project with a name that includes "Lab4" and your last name. Remember, do NOT include blanks or special characters in your project name. Then, close last week's project. Finally, create a new "GUIManager" class with a name like `IMClient` and start your work.

Grading

Completeness (14 points) / Correctness (6 points)

- GUI layout
- Connecting to the server
- Disconnecting from the server
- Sending IM messages
- Receiving IM messages
- Manual update of buddy menu
- Automatic update of buddy menu
- Correctly displaying messages sent and received

Style (10 points)

- Commenting
- Good variable names
- Good, consistent indentation
- Good use of blank lines
- Removing unused methods

Submission Instructions

Make sure to take a final look through your code checking its correctness and style. Make sure you included your name and lab section in a comment. Find the folder that BlueJ created for your project. Its name should be the one you picked for your project (something like FloydLab4).

- Click on the Desktop, then go to the "Go" menu and "Connect to Server."
- Type "cortland" in for the Server Address and click "Connect."
- Select Guest, then click "Connect."
- Select the volume "Courses" to mount and then click "OK." (and then click "OK" again)
- A Finder window will appear where you should double-click on "cs134",
- Drag your project's folder into either "Dropoff-Monday" or "Dropoff-Tuesday".

You can submit your work up to 11 p.m. two days after your lab (11 p.m. Wednesday for those in the Monday Lab, and 11 p.m. Thursday for those in the Tuesday Lab). If you submit and later discover that your submission was flawed, you can submit again. The Mac will not let you submit again unless you change the name of your folder slightly. Just add something to the folder name (like the word "revised") and the re-submission will work fine.