Chapter 19. Network Chat

Three versions of a network chat application are developed in this chapter.

1. **Threaded TCP Clients and Server.** The server uses threads to communicate with its clients, and a shared object to maintain client information. Each client employs a thread to watch for server communications (typically copies of other users' messages).
2. **UDP Multicasting Clients and a Name Server.** The clients send messages to each other via UDP multicasting. However, a client wishing to join the chat group must first communicate with a name server which sends it the group address (or rejects the request). A departing client must notify the name server, so that it can maintain a current list of participants. A client wanting a list of other users sends a message to the server, rather than querying the group.
3. **Clients using a Servlet as a Server.** The clients communicate with a chat servlet, sending messages as arguments of the servlet's URL. The servlet maintains client information, which is shared between the multiple threads executing inside the servlet instance. Each client has a separate thread which periodically queries the servlet for any new messages.

These examples illustrate the features of client/server models, peer-to-peer, and HTTP tunneling respectively, mechanisms first introduced in chapter 18.

1. Threaded TCP Clients and Server

Figure 1 shows the various objects in the threaded chat application.

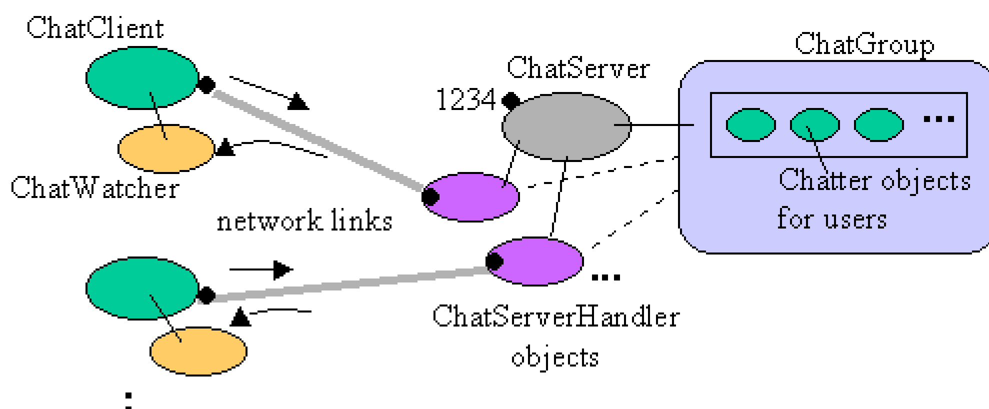

Figure 1. Objects in the Threaded Chat System.

Each client is represented by a ChatClient object and a ChatWatcher thread. ChatClient maintains the GUI, processes the user's input, and sends messages to the

 © Andrew Davison. 2003

server over a TCP link. ChatWatcher waits for messages from the server and displays them in ChatClient's GUI text area.

ChatClient can send the following messages:

- who The server returns a string containing a list of the current chat users.
- bye The client sends this message just prior to exiting.
- Any message. Other text strings sent to the server are assumed to be messages, and broadcast to all the current users.

The server can send out two types of message:

- WHO$$ cliAddr1 & port1 & ... cliAddrN & portN &
  This is sent back to a client in response to a "who" message. Each client is identified by their address and port number, which are extracted from the TCP link established when a client first connects to the server.
- (cliAddr, port): message
  This is a broadcast message, originally from the client with the specified address and port number.

Server messages are received by the ChatWatcher thread, which is monitoring incoming messages on the ChatClient's TCP link.

ChatServer is the top-level server, which spawns a ChatServerHandler thread to handle each new client. Since messages are to be transmitted between clients, the ChatServer maintains pertinent client information, accessible by all the threads. The shared data is held by a ChatGroup object, which offers synchronized methods to control the concurrency issues. A client's details, such as its input stream, is stored in a Chatter object.

Figure 2 gives the UML class diagrams for the application, showing the public methods.

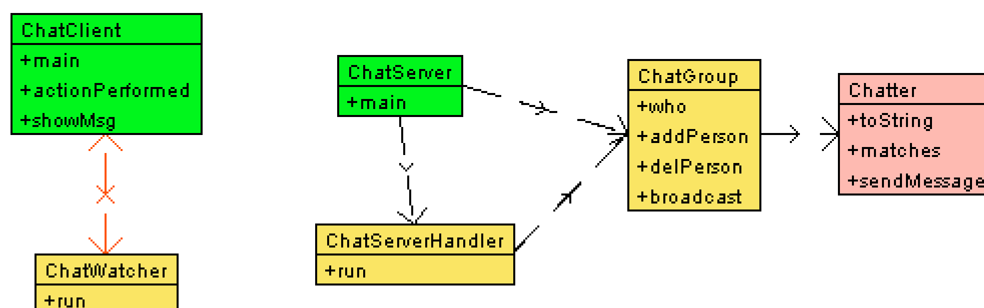

Figure 2. UML Class Diagrams for the Threaded Chat System.

 © Andrew Davison. 2003

1.1. The ChatClient Class

The GUI supported by ChatClient is shown in Figure 3.

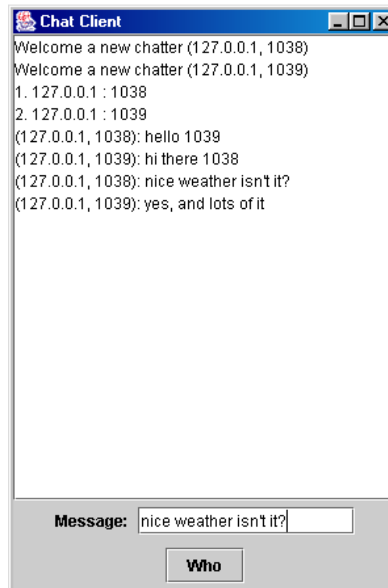

Figure 3. The ChatClient GUI.

The text area for displaying messages occupies the majority of the window. Outgoing messages are typed in a text field, and sent when the user presses enter. A 'who' message is output when the "Who" button is pressed. A 'bye' message is transmitted as a result of the user clicking on the window's close box.

The server is contacted, and the ChatWatcher thread started, in makeContact().

```
// globals
private static final int PORT = 1234;     // server details
private static final String HOST = "localhost";

private Socket sock;
private PrintWriter out;  // output to the server
    :

private void makeContact()
{
  try {
    sock = new Socket(HOST, PORT);
    BufferedReader in  = new BufferedReader(
            new InputStreamReader( sock.getInputStream() ) );
    out = new PrintWriter( sock.getOutputStream(), true );

    new ChatWatcher(this, in).start();    // watch for server msgs
  }
  catch(Exception e)
  {  System.out.println(e);  }
}
```

The output stream, out, is made global so that messages can be sent from various methods in ChatClient. However, server input is only processed by ChatWatcher, so is declared locally in makeContact() before being passed to the thread.

 © Andrew Davison. 2003

Sending messages is very easily achieved. For example, the pressing of the "who" button results in:

```
out.println("who");
```

ChatWatcher is passed a reference to ChatClient, so that it can write into the GUI's text area, jtaMesgs. This is done via the synchronized method showMsg():

```
synchronized public void showMsg(String msg)
{ jtaMesgs.append(msg);  }
```

1.2. The ChatWatcher Class

The core of the ChatWatcher class is a while loop inside run() which waits for a server message, processes it, then repeats. The two message types are the "WHO$$.." response and broadcast messages from other clients.

```
 while ((line = in.readLine()) != null) {
   if ((line.length() >= 6) &&     // "WHO$$ "
       (line.substring(0,5).equals("WHO$$")))
     showWho( line.substring(5).trim() );
          // remove WHO$$ keyword and surrounding space
   else  // show immediately
     client.showMsg(line + "\n");
 }
```

showWho() reformats the "WHO$$..." string before displaying it.

1.3. The ChatServer Class

The ChatServer constructor initializes a ChatGroup object to hold client information, and then enters a loop which deals with client connections by creating ChatServerHandler threads.

```
public ChatServer()
{ cg = new ChatGroup();
  try {
    ServerSocket serverSock = new ServerSocket(PORT);
    Socket clientSock;
    while (true) {
      System.out.println("Waiting for a client...");
      clientSock = serverSock.accept();
      new ChatServerHandler(clientSock, cg).start();
    }
  }
  catch(Exception e)
  {  System.out.println(e);  }
}
```

Each handler is given a reference to the ChatGroup object.

 © Andrew Davison. 2003

1.4. The ChatServerHandler Class

This class is very similar to the ThreadedScoreHandler class from section 4.2 (??) of chapter 18. The main differences are the calls it makes to the ChatGroup object while processing the client's messages.

The run() method sets up the input and output streams to the client, and adds (and later removes) a client from the ChatGroup object.

```
public void run()
{ try {
    // Get I/O streams from the socket
    BufferedReader in  = new BufferedReader(
          new InputStreamReader( clientSock.getInputStream() ) );
    PrintWriter out =
       new PrintWriter( clientSock.getOutputStream(), true);

    cg.addPerson(cliAddr, port, out);  // add client to ChatGroup

    processClient(in, out);            // interact with client

    // the client has finished when execution reaches here
    cg.delPerson(cliAddr, port);       // remove client details
    clientSock.close();
    System.out.println("Client (" + cliAddr + ", " +
                        port + ") connection closed\n");
  }
  catch(Exception e)
  {  System.out.println(e);  }
}
```

processClient() checks for the client's departure by looking for a "bye" message. Other messages ("who" and text messages) are passed on to doRequest().

```
private void doRequest(String line, PrintWriter out)
{ if (line.trim().toLowerCase().equals("who")) {
    System.out.println("Processing 'who'");
    out.println( cg.who() );
  }
  else  // use ChatGroup object to broadcast the message
    cg.broadcast( "("+cliAddr+", "+port+"): " + line);
}
```

1.5. The ChatGroup Class

ChatGroup handles the addition/removal of client details, the answering of "who" messages, and the broadcasting of messages to all the clients. The details are stored in an ArrayList of Chatter objects (called chatPeople), one object for each client.

A single ChatGroup object is used by all the ChatServerHandler threads, so methods which manipulate chatPeople must be synchronized.

A typical method is broadcast(), which sends a specified message to all the clients, including back to the sender.

```
synchronized public void broadcast(String msg)
```

 © Andrew Davison. 2003

```
{ Chatter c;
  for(int i=0; i < chatPeople.size(); i++) {
    c = (Chatter) chatPeople.get(i);
    c.sendMessage(msg);
  }
}  // end of broadcast()
```

1.6. The Chatter Class

The client details managed by a Chatter object are its address, port, and PrintWriter output stream. The address and port are employed to uniquely identify the client (a client has no name). The output stream is used to send messages to the client. For example:

```
private PrintWriter out;   // global
 :

public void sendMessage(String msg)
{  out.println(msg);  }
```

1.7. Discussion

A client has no name, which makes it difficult for a user to remember who is who when messages are being exchanged.

Messages can only be broadcast – there is no capability to send private messages, although that is easily fixed, as will be seen in later examples. The communication protocol is defined by the server, so it is quite simple to create a new message format, such as:

```
message / toName
```

This could be processed by the ChatServerHandler as a private message for <toName> only. There would have to be a new method in ChatGroup (e.g. void sendPrivate(String message, String toName) which would call the sendMessage() method of the Chatter object for <toName>.

Other communication patterns (e.g. periodic announcements linked to the current time) are also quite straight forward to implement.

If ChatServer fails then the entire system fails. However, the non-functioning of a single ChatServerHandler thread does not affect the others. A likely scenario is one thread being made to wait indefinitely by a client who does not communicate with it.

The main difficult of threaded servers – the sharing of data – is avoided by using a shared, synchronized ChatGroup object.

 © Andrew Davison. 2003

2. UDP Multicasting Clients and a Name Server

Figure 4 shows the various objects in the multicasting chat application.

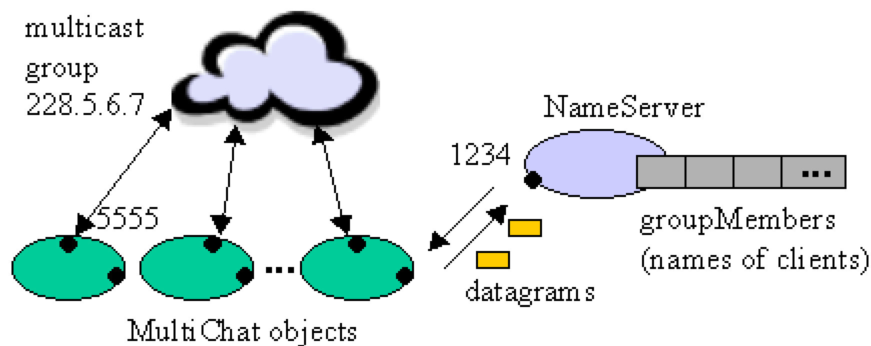

Figure 4. Multicasting Chat System.

Each user is represented by a MultiChat object which can both send and receive messages from the multicast group. However, this is not a pure peer-to-peer example since a client must first 'login' to the NameServer object and receive the address of the group. Upon leaving the group, the client must 'logout' from the server.

Each client has a name which is checked by the NameServer for uniqueness. If the name is already being used by a group member then the NameServer rejects the request to login.

"Who" messages are not multicast around the group but instead directed to the NameServer, considerably reducing the number of circulating packets.

The client / NameServer communication uses UDP, permitting the server to multiplex between client requests.

The failure of the server can influence the application, but only to the extent of preventing new clients from joining the group. Communication between existing members will be able to continue. One slight problem is that “who” messages will no longer be answered since these are processed by the server.

The messages which a client can send to the NameServer are:

- hi <client name>
  This is the ‘login’ message which either results in NameServer sending the client the multicast address (as a string), or a “no” message.
- bye <client name>
  This is the ‘logout’ message which allows the NameServer to discard the client’s name from its group members list.
- who
  The server returns the names of all the current group members.

A client can send messages to the group, for transmission to the other members. The message format:

- (<client name>): <message> [/ <toClientName>]

© Andrew Davison. 2003

The client's name is prepended to the front of the message. The message may have an optional "/" extension, which makes the message visible only to that particular client.

The UML class diagrams for the application are given in Figure 5, but are rather uninformative. Almost no public methods are required since communication is datagram-based.

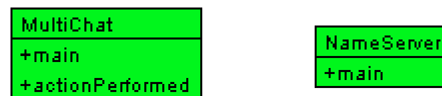

Figure 5. UML Class Diagrams for the Multicasting Chat System.

2.1. The NameServer Class

The NameServer class bears a striking resemblance at the top-level to the UDP-based server, ScoreUDPServer, in section 4.4 (??) of chapter 18.

It creates a DatagramSocket for receiving client packets, and then enters a loop waiting for them to arrive. The client's address, port, and the text of the message are extracted from an arriving packet, and the information passed to a processClient() method.

processClient() employs a four-way branch to handle the various kinds of message ("hi", "bye", "who"), along with a default catch-all branch.

```
// globals
private static final String GROUP_HOST = "228.5.6.7";
    // the multicast group address sent to new members
           :
private ArrayList groupMembers;
  // holds the names of the current members of the multicast group
           :

private void processClient(String msg, InetAddress addr, int port)
{
  if (msg.startsWith("hi")) {
    String name = msg.substring(2).trim();
    if (name != null && isUniqueName(name)) {
        groupMembers.add(name);
        sendMessage(GROUP_HOST, addr, port); // send multicast addr
    }
    else
      sendMessage("no", addr, port);
  }
  else if (msg.startsWith("bye")) {
    String name = msg.substring(3).trim();
    if (name != null)
        removeName(name);  // removes name from list
  }
  else if (msg.equals("who"))
    sendMessage( listNames(), addr, port);
  else
    System.out.println("Do not understand the message");
```

 © Andrew Davison. 2003

```
  }
```

The client's name is added to an ArrayList called groupMembers if it is not already present in the list. A "bye" message removes the name. A "who" message triggers a call to listNames() which builds a string of all the groupMembers names.

sendMessage() creates a datagram that is sent back to the client's address and port:

```
  private void sendMessage(String msg, InetAddress addr, int port)
  {
    try {
      DatagramPacket sendPacket =
          new DatagramPacket( msg.getBytes(), msg.length(),
                                          addr, port);
      serverSock.send( sendPacket );
    }
    catch(IOException ioe)
    {  System.out.println(ioe);  }
  }
```

The simplicity of the processClient() code highlights the fact that the login and logout mechanisms could be improved. There is no password associated with the name used in "hi", so a client can use any name to login. A "bye" message can come from anywhere, so it is possible for any client to logout any other. In any case, the removal of a client from the NameServer does not force a client to leave the multicast group.

The introduction of a password, together with encryption of the message inside the datagram would solve many of these problems.

The remaining concern is that the server cannot force a client to leave the multicast group. The protocol is supported at the IP level, and so is beyond the reach of the Java MuticastSocket API. Indeed, the server has no control over the group at all, aside from deciding who will be given the multicast group address. This is hardly effective since the address can be shared between clients once one user has it.

This lack of control is a crucial difference between the client/server approach and peer-to-peer, as discussed in the previous chapter.

© Andrew Davison. 2003

2.2. The MultiChat Class

A MultiChat object is capable of two forms of network communication: it can send ordinary UDP datagrams to the NameServer (and receive replies), and datagrams to the multicast group (and receive messages from the group).

A MultiChat object is invoked with the client name supplied on the command line:

```
$ java MultiChat andy
```

Its GUI is very similar to the one for the threaded chat client, as shown in Figure 6.

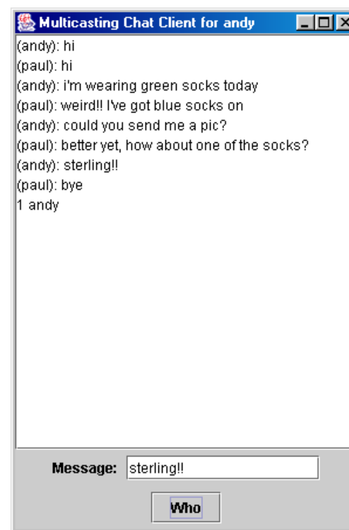

Figure 6. The Multicasting Chat Client.

One obvious difference is that messages are now prefixed with the client's name rather than an address and port.

The MultiChat constructor creates a DatagramSocket for the NameServer, and sends it a "hi <name>" message requesting the multicast group's address. If the address is sent back, the group is joined, and a "hi" message sent to it (the actual message could be anything).

```
public MultiChat(String nm)
{
   super( "Multicasting Chat Client for " + nm);
   userName = nm;
   initializeGUI();

   /* Attempt to register name and get multicast group
      address from the NameServer */
   makeClientSock();
   sendServerMessage("hi " + userName);  // say "hi" to NameServer
   checkHiResponse();

   // Connect to the multicast group; say hi to everyone
   joinChatGroup();
   sendPacket("hi");

   addWindowListener( new WindowAdapter() {
     public void windowClosing(WindowEvent e)
     { sayBye(); }   // say bye at termination time
   });
```

 © Andrew Davison. 2003

```
    setSize(300,450);
    show();

    waitForPackets();
  } // end of MultiChat();
```

2.2.1. Talking to the NameServer

makeClientSock() creates a DatagramSocket for sending and receiving communication from the server. However, it is given a timeout of five seconds:

```
      clientSock = new DatagramSocket();
      clientSock.setSoTimeout(5000);
```

A timeout is useful when the client is waiting for a datagram inside receive(). This occurs after a “hi” message has been sent, and after “who” queries. If the server has died, the timeout will limit the wait to five seconds before a SocketTimeoutException is raised.

The drawback is deciding on a reasonable timeout value. For instance, five seconds would probably be too short if the link was across the internet, through slow modems. This is why most programmers prefer to put blocking calls into separate threads so they can wait as long as they like. An alternative for TCP is to use nonblocking sockets, as illustrated in section 4.3. (??) of chapter 18.

Messages are sent to the server from sendServerMessage() by creating a datagram and sending it through clientSock.

Messages coming from the server are read by readServerMessage(). Its clientSock.receive() call is wrapped in a try-catch block in case of a timeout (or other network problems).

A call is made to readServerMessage() immediately after a message is sent to the server. There is no attempt to use a separate thread for server processing. For example, a “who” message is followed by a readServerMessage() call:

```
   sendServerMessage("who");
   String whoResponse = readServerMessage();
```

2.2.2. Talking to the Multicast Group

The multicast group address string is extracted from the “hi” response, and used by joinChatGroup() to create the multicast socket. A code fragment that does this:

```
   groupAddr = InetAddress.getByName(hiResponse);
                    :
   groupSock = new MulticastSocket(GROUP_PORT);  // port 5555
   groupSock.joinGroup(groupAddr);
```

 © Andrew Davison. 2003

sendPacket() is used to send a message to the group, and adds the "(Name):" prefix prior to delivery:

```
private void sendPacket(String msg)
{ String labelledMsg = "(" + userName + "): " + msg;
  try {
    DatagramPacket sendPacket =
      new DatagramPacket(labelledMsg.getBytes(),
             labelledMsg.length(), groupAddr, GROUP_PORT);
    groupSock.send(sendPacket);
  }
  catch(IOException ioe)
  {  System.out.println(ioe);  }
}
```

Text messages are sent to the group when the user hits enter in the messages text field. actionPerformed() calls sendMessage():

```
public void actionPerformed(ActionEvent e)
{ if (e.getSource() == jbWho)
    doWho();
  else if (e.getSource() == jtfMsg)
    sendMessage();
}
```

sendMessage() extracts the string from the text field, checks it for validity, and passes it to sendPacket(). All of this is done inside the GUI thread of the MultiChat application.

Multicast messages may arrive at any time, and so should be handled in a separate thread. Instead of creating a new Thread object, MultiChat uses the application thread by entering an infinite loop inside waitForPackets().

```
private void waitForPackets()
{ DatagramPacket packet;
  byte data[];

  try {
    while (true) {
      data = new byte[PACKET_SIZE];    // set up an empty packet
      packet = new DatagramPacket(data, data.length);
      groupSock.receive(packet);       // wait for a packet
      processPacket(packet);
    }
  }
  catch(IOException ioe)
  {  System.out.println(ioe);  }
}
```

This is the same approach as in the ScoreUDPClient of section 4.4. (??) of chapter 18.

processPacket() extracts the text from the incoming packet, and displays it (if it is visible to this client):

 © Andrew Davison. 2003

```
private void processPacket(DatagramPacket dp)
{
  String msg = new String( dp.getData(), 0, dp.getLength() );
  if (isVisibleMsg(msg, userName))
    showMsg(msg + "\n");
}
```

A message is visible if it has no "/ <name>" part, or "/ <name>" contains the client's name, or the message is originally from the client. This latter condition can be checked by looking at the start of the message which has the sender's name in brackets.

2.3. How Invisible are Invisible Messages?

A message will not appear in a client's text display area if it has a "/name" extension that refers to a different client. Is this sufficient for private communication between two users? Unfortunately no, since the message is still being multicast to every client. This means that a hacker could modify the MultiChat code to display everything that arrived. Privacy requires that the messages be encrypted.

The other misleading thing about private communication is that it appears to be point-to-point between users, and so seemingly more efficient than multicasting. Unfortunately, this is also not true. True point-to-point communication would require another communication link, perhaps utilizing the DatagramSockets utilized for client/server interaction.

2.4. Ways to Implement "who"

MultiChat handles a "who" message by querying the server for a list of names. Another approach would be to multicast the message to all the clients to gather responses. The main benefit of this would be the absence of a server, which can fail or otherwise stop responding.

The reason that multicasting is not used for "who" is the large number of messages it would add to the system. The original "who" query would cause n messages to arrive at the n clients. Each client would send a reply, multicast to every other client, not just the one interested in the answer. This would cause a total of n*n messages to circulate through the group. The grand total for a single "who" query is $n + n^2$. This should be compared with the server approach which only generates two messages: the request and the server's reply.

This shows that alternatives to multicasting should be explored where possible, since bandwidth is such a crucial commodity.

© Andrew Davison. 2003

3. Clients using a Servlet as a Server

Figure 7 shows the various objects in the servlet-based chat application.

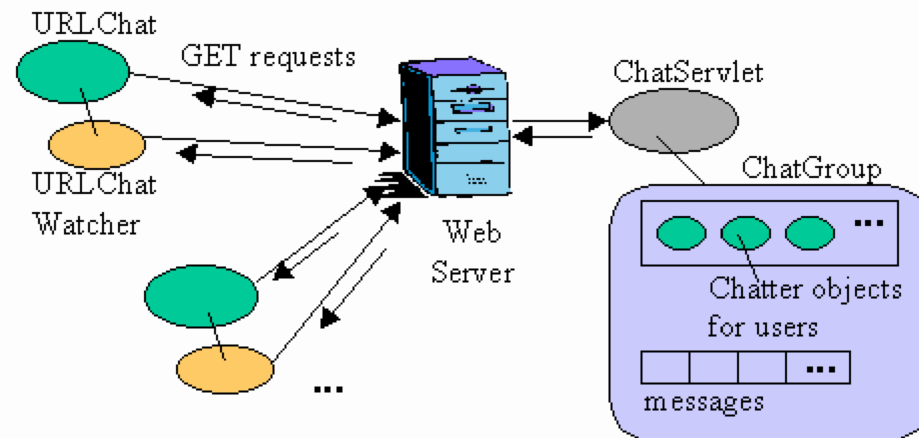

Figure 7. The Servlet-based Chat System.

Each client is represented by two objects: a URLChat object manages the GUI and translates user input into messages for the Web server. The URLChatWatcher thread periodically queries the server for new messages sent by the other users.

The communication is implemented by sending the server a GET request for the ChatServlet object, with message details added as arguments to the URL. A client is identified by a name and a cookie. The cookie is presented to a user when he/she sends a "hi" message to the servlet upon joining the chat system. Most messages require the name and cookie information, which are used to verify the identity of the message sender.

The servlet maintains a synchronized ChatGroup object, which bears many similarities to the ChatGroup object used in the threaded chat application. Client information is held in Chatter objects, while client messages are stored in a messages ArrayList.

A key difference between ChatServlet and the server in the first chat example, is that ChatServlet cannot initiate communication with its clients. It must wait for a URLChatWatcher thread to ask for messages before it can send them out.

URLChat transmits its messages as arguments attached to the URL http://localhost:8080/servlet/ChatServlet. The notation for an argument in a GET request is "name=value", with multiple name/value pairs separated by '&'. The parameters follow a "?' after the URL.

The four possible messages types are:

- ChatServlet?cmd=**hi**&name=??. This is a "hi" message, used by a client to ask for chat group membership. The name parameter value (represented by '??') holds the client's name. The servlet returns a cookie containing a user ID (uid), or rejects the client.
- ChatServlet?cmd=**bye**&name=?? + uid cookie. This is the "bye message", which signals the client's departure. The cookie is included in the header part of the GET message, not as a URL argument.

 © Andrew Davison. 2003

- ChatServlet?cmd=**who**. The "who" message requires no client name or cookie parameters. The servlet returns the names of the clients currently using the application.
- ChatServlet?cmd=**msg**&name=??&msg=?? + uid cookie. This message sends chat text to the servlet as the 'msg' parameter value. The string is added to the servlet's messages list, but only if the client name and uid are correct.

The chat message format is the same as in the multicasting example: if the text has a "/ toName" extension then it is intended only for the client with that name.

URLChatWatcher periodically sends a "read" message:

- ChatServlet?cmd=**read**&name=?? + uid cookie. This retrieves all the *visible* chat messages stored by the servlet since the last read. A visible message is one intended for everyone (the default), or one with a "/ toName" extension which matches this client's name.

The idea behind the cookie is to act as an additional form of identification, paired with the client's name. However, cookies are not passwords, being passed in plain text form in the headers of the messages. To act as a password, it would be necessary to add encryption to the interaction, possibly by using HTTPS rather than HTTP.

Figure 8 gives the UML class diagrams for the application, showing the public methods.

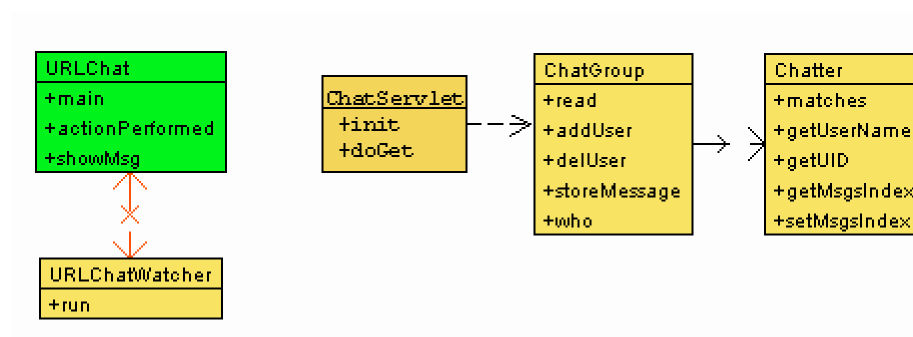

Figure 8. UML Class Diagrams for the Servlet-based Chat System.

 © Andrew Davison. 2003

3.1. The URLChat Class

Figure 9 shows the GUI for URLChat, which is identical to the earlier examples. A "hi" message is generated when the client is first invoked, a "bye" message is sent when the user clicks on the window's close box, a "who" message is transmitted when the "Who" button is pressed, and the entering of a string in the text field results in a "msg" chat message.

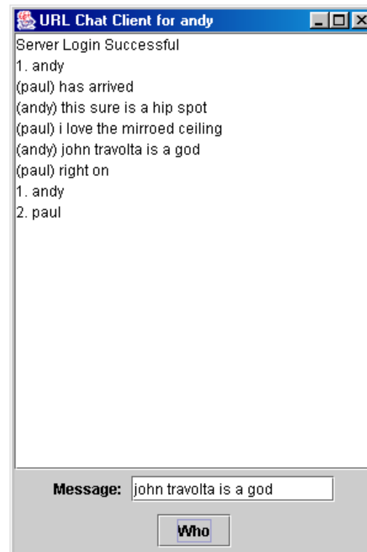

Figure 9. The URLChat GUI.

The constructor carries out several tasks: it creates the GUI, sets timeout properties, sends a "hi" message, invokes a URLChatWatcher, and links the "bye" output to the close box:

```
// globals
private String userName;  // for this client
private String cookieStr = null;
     :

public URLChat(String nm)
{  super( "URL Chat Client for "+ nm);
   userName = nm;
   initializeGUI();

   // set the properties used for URL timeouts (in ms)
   Properties props = System.getProperties();
   props.put("sun.net.client.defaultConnectTimeout", "2000");
   props.put("sun.net.client.defaultReadTimeout", "2000");
   System.setProperties(props);

   sayHi();
   new URLChatWatcher(this, userName, cookieStr).start();

   addWindowListener( new WindowAdapter() {
     public void windowClosing(WindowEvent e)
     { sayBye(); }
   });

   setSize(300,450);
   show();
```

 © Andrew Davison. 2003

```
  } // end of URLChat();
```

A client utilizes URL GET requests to communicate with the servlet. If the server is down, or the servlet is unavailable, then the client must wait for the URL request to timeout. This often amounts to several minutes delay before an exception is raised. Fortunately, J2SDK 1.4 introduced two properties for adjusting the default connection timeout and the reading timeout (how long to wait when the reading of a Web page is delayed). These are associated with every URLConnection object created in the application. The main problem with timeouts, as always, is deciding on a reasonable value which takes network latency into account.

Another approach is to wrap each connection in a thread using Thread.join() as a timeout mechanism. This has the advantage that different timeout values can be assigned to different URLs.

sayHi() illustrates how a GET request is constructed using a URL object. The response from the servlet may include a "Set-Cookie" header – this is the usual way that a cookie is sent to a browser. "Set-Cookie" is not a part of the HTTP standard, but is used by all Web servers and browsers. The need to read the response page's headers requires a URLConnection object.

The returned page is plain text, consisting of a single line containing "ok" or "no".

```
  private void sayHi()
  {
    try {
      URL url  = new URL(SERVER + "?cmd=hi&name=" +
                  URLEncoder.encode(userName, "UTF-8") );

      URLConnection conn = url.openConnection();
      cookieStr = conn.getHeaderField("Set-Cookie");  // get cookie

      System.out.println("Received cookie: " + cookieStr);
      if (cookieStr != null) {
        int index = cookieStr.indexOf(";");
        if (index != -1)
          cookieStr = cookieStr.substring(0, index);  //remove extras
      }

      BufferedReader br = new BufferedReader(
              new InputStreamReader( conn.getInputStream() ));
      String response = br.readLine().trim();
      br.close();

      if (response.equals("ok") && (cookieStr != null))
        showMsg("Server Login Successful\n");
      else {
        System.out.println("Server Rejected Login");
        System.exit(0);
      }
    }
    catch(Exception e)
    { System.out.println(e);
      System.exit(0);
    }
```

```
  } // end of sayHi()
```

The client name must be URL-encoded prior to transmission. Essentially this replaces any spaces in the string by '+'s, and alters certain other characters to a hexadecimal format.

The value of a "Set-Cookie" header is a string containing the cookie value separated from various other things by a ";" (these may include a comment, path and domain qualifiers, a maximum age, and a version number). This additional information is not required, so sayHi() strips it away, before storing the string in the cookieStr global. In fact, the servlet creates a cookie holding only a value, so the editing code is not really necessary in this example.

The client will terminate if the response to the "hi" message is "no", or the cookie string is not initialized. All further communication with the servlet (aside from "who" messages) requires the cookie value.

If a servlet (or its Web server) is unavailable then the URL request will timeout after two seconds (due to the timeout properties set in the constructor), and raise an exception. This allows a client to respond to the server's absence by exiting.

The showMsg() method writes into the GUI's text area. showMsg() is synchronized since it may also be called by the URLChatWatcher thread.

```
  synchronized public void showMsg(String msg)
  { jtaMesgs.append(msg);  }
```

sendURLMessage() sends a chat message to the servlet. It is quite similar to sayHi() except that it adds the cookie value to the GET request, by including a "Cookie" header in the output.

The response page returned by the servlet is a single line containing "ok" or "no".

```
  private void sendURLMessage(String msg)
  {
    try {
      URL url  = new URL(SERVER + "?cmd=msg&name=" +
                  URLEncoder.encode(userName, "UTF-8") +
                  "&msg=" + URLEncoder.encode(msg, "UTF-8") );
      URLConnection conn = url.openConnection();

      conn.setRequestProperty("Cookie", cookieStr); // add cookie

      BufferedReader br = new BufferedReader(
              new InputStreamReader( conn.getInputStream() ));
      String response = br.readLine().trim();
      br.close();

      if (!response.equals("ok"))
        showMsg("Message Send Rejected\n");
      else // display message immediately
        showMsg("(" + userName + ") " + msg + "\n");
    }
    catch(Exception e)
    { showMsg("Servlet Error. Did not send: " + msg + "\n");
```


© Andrew Davison. 2003

```
      System.out.println(e);
    }
  }  // end of sendURLMessage()
```

3.1.1. Avoiding Delays

A small but significant part of sendURLMessage() is the call to showMsg() to display the output message if the servlet's response was "ok". This means that the message appears in the client's text area almost as soon as it is typed.

The alternative would be to wait until it was retrieved, together with the other recent messages, by URLChatWatcher. The drawback of this approach is the delay before the user sees a visual conformation that their message has been sent. This delay depends on the frequency of URLChatWatcher's requests to the servlet (currently every two seconds). A message echoing delay like this gives an impression of execution slowness and communication latency, which is best avoided.

The drawback of the current design (immediate echoing of the output message) is that the user's contribution to a conversation may not appear on screen in the order that the conversation is recorded in the servlet.

3.2. The URLChatWatcher Class

URLChatWatcher starts an infinite loop which sleeps for a while (two seconds), then sends a "read" request for new chat messages. The answer is processed, displayed in the client's text area, and the watcher repeats.

```
  public void run()
  { URL url;
    URLConnection conn;
    BufferedReader br;
    String line, response;
    StringBuffer resp;

    try {
      String readRequest = SERVER + "?cmd=read&name=" +
                  URLEncoder.encode(userName, "UTF-8") ;
      while(true) {
        Thread.sleep(SLEEP_TIME);  // sleep for 2 secs

        url  = new URL(readRequest);   // send a "read" message
        conn = url.openConnection();

        // Set the cookie value to send
        conn.setRequestProperty("Cookie", cookieStr);

        br = new BufferedReader(
              new InputStreamReader( conn.getInputStream() ));
        resp = new StringBuffer();    // build up the response
        while ((line = br.readLine()) != null) {
          if (!fromClient(line))   // if not from client
            resp.append(line+"\n");
        }
        br.close();
```

 © Andrew Davison. 2003

```
        response = resp.toString();
        if ((response != null) && !response.equals("\n"))
          client.showMsg(response);    // show the response
      }
    }
    catch(Exception e)
    { client.showMsg("Servlet Error: watching terminated\n");
      System.out.println(e);
    }
  } // end of run()
```

The try-catch block allows the watcher to respond to the server's absence by issuing an error message before terminating.

The response page may contain multiple lines of text, one line for each chat message. The watcher filters out chat message originating from its client, since these will have already been printed when they were sent out.

A drawback of this filtering is that when a client joins the chat system again in the future, messages stored from a prior visit will not be shown in the display text area. This can be remedied by making the fromClient() test a little more complicated than the existing code.

```
  private boolean fromClient(String line)
  { if (line.startsWith("("+userName))
      return true;
    return false;
  }
```

3.3. The ChatServlet Class

The initialization phase of a ChatServlet object creates a new ChatGroup object for holding client details. Its doGet() method tests the "cmd" parameter of GET requests to decide what message is being received.

```
   // globals
   private ChatGroup cg;   // for storing client information

   public void init() throws ServletException
   {  cg = new ChatGroup();  }


   public void doGet( HttpServletRequest request,
                      HttpServletResponse response )
      throws ServletException, IOException
   // look at cmd parameter to decide which message the client sent
   {
     String command = request.getParameter("cmd");
     System.out.println("Command: " + command);

     if (command.equals("hi"))
       processHi(request, response);
     else if (command.equals("bye"))
       processBye(request, response);
     else if (command.equals("who"))
       processWho(response);
     else if (command.equals("msg"))
       processMsg(request, response);
```

```
    else if (command.equals("read"))
      processRead(request, response);
    else
      System.out.println("Did not understand command: " + command);
  }  // end of doGet()
```

An understanding of this code requires a knowledge of the servlet lifecycle. A Web server will typically create a single servlet instance, resulting in init() being called once. However, each GET request will usually be processed by spawning a separate thread which executes doGet() for itself. This is quite similar to the execution pattern employed by the threaded server in the first chat example.

The consequence of this approach is that data which may be manipulated by multiple doGet() threads must be synchronized. The simplest solution is to place this data in an object that is only accessible by synchronized methods. Consequently, the single ChatGroup object (created in init()) holds client details and the list of chat messages.

processHi() deals with a message with the format “ChatServlet?cmd=hi&name=??”. The name parameter must be extracted, tested, and a cookie created for the client’s new user ID. The cookie is sent back as a header, and the response page is a single line containing “ok”, or “no” if something is wrong.

```
private void processHi(HttpServletRequest request,
                       HttpServletResponse response)
  throws IOException
{
  int uid = -1;  // default for failure
  String userName = request.getParameter("name");

  if (userName != null)
    uid = cg.addUser(userName);  // attempt to add to group

  if (uid != -1) {  // the request has been accepted
    Cookie c = new Cookie("uid", ""+uid);
    response.addCookie(c);
  }

  PrintWriter output = response.getWriter();
  if (uid != -1)
    output.println("ok");
  else
    output.println("no");  // request was rejected
  output.close();
}  // end of processHi()
```

The cookie is created using the Cookie class, and added to the headers of the response with the addCookie() method. The actual cookie header has the format:

```
Set-Cookie: uid= <some number>
```

The UID number is generated when ChatGroup creates a Chatter object for the new client. It may have any value between 0 and 1024.

If the UID value returned by ChatGroup is -1 then the membership request has been rejected, because the client’s name is already being used by another person.

processMsg() is typically of the other processing methods since it deals with an incoming cookie in the header of the request. It handles a message with the format:

```
ChatServlet?cmd=msg&name=??&msg=?? + uid cookie
```

It attempts to add the chat message to the messages list maintained by ChatGroup.

```
private void processMsg(HttpServletRequest request,
                        HttpServletResponse response)
  throws IOException
{
  boolean isStored = false;   // default for failure
  String userName = request.getParameter("name");
  String msg = request.getParameter("msg");

  System.out.println("msg: " + msg);

  if ((userName != null) && (msg != null)) {
    int uid = getUidFromCookie(request);
    isStored = cg.storeMessage(userName,uid,msg); //add msg to list
  }

  PrintWriter output = response.getWriter();
  if (isStored)
    output.println("ok");
  else
    output.println("no");   // something wrong
  output.close();
}  // end of processBye()
```

Processing is aborted if the name or chat message are missing. The call to getUidFromCookie() will return a number (which may be -1, signifying an error). The ChatGroup object is then given the task of storing the chat message, which it only does if the name/uid pair match an existing client. The resulting isStored value is employed to decide on the response page sent back to the client.

getUidFromCookie() must deal with the possibility of multiple cookies in the request, and so has to search for the one with the “uid” name. A missing “uid” cookie, or one with a non-numerical value, causes a -1 to be returned.

```
private int getUidFromCookie(HttpServletRequest request)
{
  Cookie[] cookies = request.getCookies();
  Cookie c;
  for(int i=0; i < cookies.length; i++) {
    c = cookies[i];
    if (c.getName().equals("uid")) {
      try {
        return Integer.parseInt( c.getValue() );
      }
      catch (Exception ex){
        System.out.println(ex);
        return -1;
      }
    }
  }
  return -1;
```


© Andrew Davison. 2003

```
} // end of getUidFromCookie()
```

The getCookies() method returns an array of Cookie objects, which can be examined with getName() and getValue().

3.4. The ChatGroup Class

ChatGroup maintains two ArrayLists: chatUsers and messages. chatUsers is an ArrayList of Chatter objects; each Chatter object stores a client's name, UID, and the number of messages read by its CharURLWatcher thread. messages is an ArrayList of strings (one for each chat message).

All the public methods are synchronized since there may be many doGet() threads competing to access the ChatGroup object at the same time.

The messages ArrayList is a very simple solution to the problem of storing chat messages. The drawback is that the list will grow very long as the volume of communication increases. However, the messages list is cleared when there are no users in the chat group.

Another optimization, which is not utilized, is to delete messages which have been read by all the current users. This would keep the list small, since each client's URLChatWatcher reads the list every few seconds. The drawback is the extra complexity of adjusting the list and the Chatter objects' references to it.

A possible advantage of maintaining a lengthy messages list is that new members get to see earlier conversations when they first join the system. The list is sent to a new client when its watcher thread sends its first "read" message.

Another approach might be to archive older messages in a text file. This could be accessed when required without being a constant part of the messages list. It could also be employed as a backup mechanism in case of server failure.

addUser() adds a new user to the ChatGroup, but only if their name is unique. A UID for the user is returned.

```
// globals
private ArrayList chatUsers;
private ArrayList messages;
private int numUsers;
   :

synchronized public int addUser(String name)
// adds a user, returns UID if okay, -1 otherwise
{
  if (numUsers == 0)   // no one logged in
    messages.clear();

  if (isUniqueName(name)) {
    Chatter c = new Chatter(name);
    chatUsers.add(c);
    messages.add("(" + name + ") has arrived");
    numUsers++;
    return c.getUID();
  }
```

 © Andrew Davison. 2003

```
    return -1;
  }
```

One of the more complicated methods in ChatGroup is read(), which is called when a watcher thread requires new messages.

```
  synchronized public String read(String name, int uid)
  {
    StringBuffer msgs = new StringBuffer();
    Chatter c = findUser(name, uid);

    if (c != null) {
      int msgsIndex = c.getMsgsIndex();  // where read to last time
      String msg;
      for(int i=msgsIndex; i < messages.size(); i++) {
        msg = (String) messages.get(i);
        if (isVisibleMsg(msg, name))
          msgs.append( msg + "\n" );
      }
      c.setMsgsIndex( messages.size());  //update client's read index
    }
    return msgs.toString();
  }
```

The method begins by calling findUser() to retrieve the Chatter object for the given name and uid, which may return null if there is no match.

Since the messages list retains all the chat messages, it is necessary for each Chatter object to record the number of messages already read. This is retrieved with a call to getMsgsIndex(), and updated with setMsgsIndex(). This number corresponds to an index into the messages list, and so can be used to initialize the for-loop that collects the chat messages.

If the message list was periodically purged of messages that had been read by all the current users, then it would be necessary to adjust the index number stored in each Chatter object.

Only visible messages are returned – those without a “/ toName” extension, or an extension which names the client.

3.5. The Chatter Class

Each Chatter object stores the client's name, their UID, and the current number of messages read from the chat messages list.

The UID is generated in a trivial manner by generating a random number between 0 and ID_MAX (1024). The choice of ID_MAX has no significance.

```
    uid = (int) Math.round( Math.random()* ID_MAX);
```

 © Andrew Davison. 2003