

```
1 package server;
2
3 import java.io.*;
4 import java.util.*;
5 import ocsf.server.*;
6 import common.*;
7
8 public class EchoServer implements Observer {
9
10     final public static int DEFAULT_PORT = 5555;
11
12     static final String PASSWORDFILE = "passwords.txt";
13
14     static final int LINEBREAK = 10; // Added in Phase 3
15     static final int RETURN = 13; // Added in Phase 3
16     static final int SPACE = 32; // Added in Phase 3
17
18     ObservableOriginatorServer server;
19
20     String serverChannel = null;
21
22     Vector blockedUsers = new Vector();
23
24     private ChatIF serverUI;
25
26     private boolean closing = false;
27
28     public EchoServer(ObservableOriginatorServer server, ChatIF serverUI) throws
IOException {
29         this.server= server;
30         this.serverUI = serverUI;
31         server.addObserver(this);
32         server.listen();
33     }
34
35     public void sendToAllClients(Object msg) {
36         Thread[] clients = server.getClientConnections();
37
38         for (int i = 0; i < clients.length; i++) {
39             ConnectionToClient c = (ConnectionToClient)(clients[i]);
40
41             try {
42                 //If the client is logged on, send the message
43                 if (((Boolean)(c.getInfo("passwordVerified"))).booleanValue())
44                     c.sendToClient(msg);
45             } catch (IOException e) {
46                 serverUI.display("WARNING - Cannot send message to a client.");
47             }
48         }
49     }
50
51     public void update(Observable obs, Object msg) {
52         // block added in phase 4 to handle Originator Messages.
53         if (!(msg instanceof OriginatorMessage))
54             return;
55
56         OriginatorMessage message= (OriginatorMessage)msg;
57
58         if (!(message.getMessage() instanceof String))
```

```

59     return;
60
61     String command = (String)message.getMessage();
62     ConnectionToClient client= (ConnectionToClient)message.getOriginator();
63
64     if (command.startsWith(ObservableServer.CLIENT_CONNECTED)) {
65         clientConnected(client);
66         return;
67     } else if (command.startsWith(ObservableServer.CLIENT_DISCONNECTED)) {
68         clientDisconnected(client);
69         return;
70     } else if (command.startsWith(ObservableServer.CLIENT_EXCEPTION)) {
71         int ie= command.indexOf('.');
72         clientException(client, new Exception(command.substring(ie)));
73         return;
74     } else if (command.startsWith(ObservableServer.LISTENING_EXCEPTION)) {
75         int ie= command.indexOf('.');
76         listeningException(new Exception(command.substring(ie)));
77         return;
78     } else if (command.startsWith(ObservableServer.SERVER_STARTED)) {
79         serverStarted();
80         return;
81     } else if (command.startsWith(ObservableServer.SERVER_STOPPED)) {
82         serverStopped();
83         return;
84     } else if (command.startsWith(ObservableServer.SERVER_CLOSED)) {
85         serverClosed();
86         return;
87     }
88
89     // In general, display the command on the server's UI
90     // Don't display it if the user is blocked
91     if (!blockedUsers.contains(((String)(client.getInfo("loginID"))))) {
92         // Only display it if the server is on the same channel as
93         // the client or is in the 'superchannel'.
94         // The server is in the superchannel by default, and this is
95         // indicated by serverChannel being null.
96         if (serverChannel == null || serverChannel.equals(client.getInfo("channel")))
97     {
98         serverUI.display("Message: \"\" + command + "\" from \" +
client.getInfo("loginID"));
99     }
100
101     // If the user has logged in, process the command or send the message
102     if (((Boolean)(client.getInfo("passwordVerified"))).booleanValue()) {
103         // If the command was to list the users. Added in phase 3.
104         if (command.startsWith("#whoison"))
105             sendListOfClients(client);
106
107         // If the command was to retrieve the channel. Added phase 3
108         if (command.startsWith("#getchannel")) {
109             try {
110                 client.sendToClient("Currently on channel: \" + client.getInfo("channel"));
111             } catch(IOException e) {
112                 serverUI.display("Warning: Error sending message.");
113             }
114         }
115
116         // If the command was to send a private message. Added phase 3.

```

```
117     if (command.startsWith("#private"))
118         handleCmdPrivate(command, client);
119
120     // If the command was to change channels. Added phase 3.
121     if (command.startsWith("#channel"))
122         handleCmdChannel(command, client);
123
124     // If the command was to return to the main channel. Added phase 3.
125     if (command.startsWith("#nochannel"))
126         handleCmdChannel("#channel main", client);
127
128     // If the command was to broadcast a public message. Added phase 3.
129     if (command.startsWith("#pub"))
130         handleCmdPub(command, client);
131
132     // If the command was to forward messages. Added phase 3.
133     if (command.startsWith("#fwd"))
134         handleCmdFwd(command, client);
135
136     // If the command was to stop forwarding messages. Added phase 3.
137     if (command.startsWith("#unfwd")) {
138         client.setInfo("fwdClient", "");
139
140         try {
141             client.sendToClient("Messages will no longer be forwarded");
142         } catch (IOException e) {
143             serverUI.display("Warning: Error sending message.");
144         }
145     }
146
147     // If the command was to block a user. Added phase 3.
148     if (command.startsWith("#block"))
149         handleCmdBlock(command, client);
150
151     // If the command was to unblock users. Added phase 3.
152     if (command.startsWith("#unblock"))
153         handleCmdUnblock(command, client);
154
155     // If the command was to verify the users a client blocks. Added phase 3.
156     if (command.startsWith("#whoiblock"))
157         handleCmdWhoiblock(client);
158
159     // If the command was to verify the users who are blocking
160     // the client requesting the check. Added phase 3.
161     if (command.startsWith("#whoblocksme"))
162         checkForBlocks((String)(client.getInfo("loginID")), client);
163
164     // If no command is recognized, send a message to the client's current
165     channel.
166     if (!command.startsWith("#")) {
167         sendChannelMessage(client.getInfo("loginID") + "> " + command,
168             (String)client.getInfo("channel"),
169             (String)(client.getInfo("loginID")));
170     }
171     //If the user is not logged in, log him in.
172     else {
173         clientLoggingIn(command, client);
174     }
175 }
```

```
176
177 /**
178  * This method is called to handle data entered from the Server's console.
179  * @param message The message typed by the user.
180  */
181 public synchronized void handleMessageFromServerUI(String message) {
182     //If the command is #quit. Added in phase 2
183     if (message.startsWith("#quit"))
184         quit();
185
186     //If the command is #stop. Added in phase 2
187     if (message.startsWith("#stop")) {
188         if(server.isListening()) {
189             server.stopListening();
190         } else {
191             serverUI.display("Cannot stop the server before it is restarted.");
192         }
193
194         return;
195     }
196
197     //If the command is #start. Added in phase 2
198     if (message.startsWith("#start")) {
199         closing = false;
200         if (!server.isListening()) {
201             try {
202                 server.listen();
203                 serverChannel = null;
204             } catch (IOException e) {
205                 serverUI.display("Cannot listen. Terminating server.");
206                 quit();
207             }
208         } else {
209             serverUI.display("Server is already running.");
210         }
211         return;
212     }
213
214     //If the command is #close. Added in phase 2
215     if (message.startsWith("#close")) {
216         closing = true; // Indicates server is closing down
217         sendToAllClients("Server is shutting down.");
218         sendToAllClients("You will be disconnected.");
219
220         try {
221             server.close();
222         } catch (IOException e) {
223             serverUI.display("Cannot close normally. Terminating server.");
224             quit();
225         }
226         return;
227     }
228
229     //If the command is #getport. Added in phase 2
230     if (message.startsWith("#getport")) {
231         serverUI.display("Current port: " + server.getPort());
232         return;
233     }
234
235     //If the command is #setport. Added in phase 2
```

```
236     if (message.startsWith("#setport")) {
237         if ((server.getNumberOfClients() != 0) || (server.isListening())) {
238             serverUI.display("Cannot change port while clients are "
239                             + "connected or while server is listening.");
240         } else {
241             try {
242                 int port = 0;
243                 port = Integer.parseInt(message.substring(9));
244
245                 //If the port number is invalid
246                 if ((port < 1024) || (port > 65535)) {
247                     server.setPort(5555);
248                     serverUI.display("Invalid port number. Port unchanged.");
249                 } else {
250                     server.setPort(port);
251                     serverUI.display("Port set to " + port);
252                 }
253             } catch (Exception e) {
254                 serverUI.display("Invalid use of the #setport command.");
255                 serverUI.display("Port unchanged.");
256             }
257         }
258         return;
259     }
260
261     //If command is #whoison (List users) Added in phase 3.
262     if (message.startsWith("#whoison")) {
263         sendListOfClients(null);
264         return;
265     }
266
267     //If the command was a punt command (boot user) Added in phase 3.
268     if (message.startsWith("#punt")) {
269         handleServerCmdPunt(message);
270         return;
271     }
272
273     //If message is a #warn command. Added in phase 3.
274     if (message.startsWith("#warn")) {
275         handleServerCmdWarn(message);
276         return;
277     }
278
279     //If command is #channel. Added in phase 3
280     if (message.startsWith("#channel")) {
281         String oldChannel = serverChannel;
282         if (!(oldChannel == null)) {
283             sendChannelMessage("The server has left this channel.", serverChannel, "");
284         }
285
286         try {
287             serverChannel = message.substring(9);
288         } catch (StringIndexOutOfBoundsException e) {
289             serverChannel = null;
290             serverUI.display("Server will now receive all messages.");
291         }
292
293         if (serverChannel != null) {
294             sendChannelMessage("The server has joined this channel.", serverChannel,
295 ""));
```

```
295     }
296
297     serverUI.display("Now on channel: " + serverChannel);
298     return;
299 }
300
301 //If command is #nochannel. Added in phase 3.
302 if (message.startsWith("#nochannel")) {
303     if (serverChannel != null) {
304         sendChannelMessage("The server has left this channel.", serverChannel, "");
305     }
306
307     serverChannel = null;
308     serverUI.display("Server will now receive all messages.");
309     return;
310 }
311
312 //If command is #pub. Added in phase 3.
313 if (message.startsWith("#pub")) {
314     handleCmdPub(message, null);
315     return;
316 }
317
318 //If command is #getchannel
319 if (message.startsWith("#getchannel")) {
320     if (server.isListening() || server.getNumberOfClients() > 0) {
321         serverUI.display("Currently on channel: " + serverChannel);
322     } else {
323         serverUI.display("Server has no active channels.");
324     }
325     return;
326 }
327
328 //If the command is to block a user. Added in phase 3.
329 if (message.startsWith("#block")) {
330     handleServerCmdBlock(message);
331     return;
332 }
333
334 //If the command was to unblock. Added in phase 3.
335 if (message.startsWith("#unblock")) {
336     handleCmdUnblock(message, null);
337     return;
338 }
339
340 //If the command is to check which users are blocked. Added in phase 3.
341 if (message.startsWith("#whoiblock")) {
342     handleCmdWhoiblock(null);
343     return;
344 }
345
346 //If command to send a private message. Added in phase 3.
347 if (message.startsWith("#private")) {
348     handleCmdPrivate(message, null);
349     return;
350 }
351
352 //If command is to check users who are blocking the server. Added phase 3.
353 if (message.startsWith("#whoblocksme")) {
354     checkForBlocks("server", null);
```

```

355     return;
356 }
357
358 //If command is a help command.
359 if (message.startsWith("#?") || message.startsWith("#help")) {
360     serverUI.display("\nServer-side command list:"
361         + "\n#block <loginID> -- Blocks all messages from the specified client."
362         + "\n#channel <channel> -- Connects to the specified channel."
363         + "\n#close -- Stops the server and disconnects all users."
364         + "\n#getchannel -- Gets the channel the server is currently connected to."
365         + "\n#getport -- Gets the port the server is listening on."
366         + "\n#help OR #? -- Lists all commands and their use."
367         + "\n#nochannel -- Returns the server to the super-channel."
368         + "\n#private <loginID> <msg> -- Sends a private message to the specified
client."
369         + "\n#pub -- Sends a public message."
370         + "\n#punt <loginID> -- Kicks client out of the chatroom."
371         + "\n#quit -- Terminates the server and disconnects all clients."
372         + "\n#setport <newport> -- Specify the port the server will listen on."
373         + "\n#start -- Makes the server restart accepting connections."
374         + "\n#stop -- Makes the server stop accepting new connections."
375         + "\n#unblock -- Unblock messages from all blocked clients."
376         + "\n#unblock <loginID> -- Unblock messages from the specified client."
377         + "\n#warn <loginID> -- Sends a warning message to the specified client."
378         + "\n#whoblockme -- List clients who are blocking messages from the server."
379         + "\n#whoiblock -- List all clients that the server is blocking messages
from."
380         + "\n#whoison -- Gets a list of all users and channel they are connected
to.");
381     return;
382 }
383
384 //If not a server-side command or is a message is to be displayed
385 if (!(message.startsWith("#"))) {
386     serverUI.display("SERVER MESSAGE> " + message);
387     sendChannelMessage("SERVER MESSAGE> " + message, (serverChannel == null ?
"main" : serverChannel), "server");
388 } else {
389     serverUI.display("Invalid command.");
390 }
391 }
392
393 /**
394  * This method gracefully kills the server.
395  */
396 public void quit() {
397     try {
398         closing = true;
399         sendToAllClients("Server is quitting.");
400         sendToAllClients("You will be disconnected.");
401         server.close();
402     } catch (IOException e) {}
403     System.exit(0);
404 }
405
406 /**
407  * This method overrides the one in the superclass. Called
408  * when the server starts listening for connections.
409  */
410 protected void serverStarted() {

```



```
411     if (server.getNumberOfClients() != 0)
412         sendToAllClients("Server has restarted accepting connections.");
413
414     serverUI.display("Server listening for connections on port " +
server.getPort());
415 }
416
417 /**
418  * This method overrides the one in the superclass. Called
419  * when the server stops listening for connections.
420  */
421 protected void serverStopped() {
422     serverUI.display("Server has stopped listening for connections.");
423
424     // If server is closing, the clients have already been notified.
425     if (!closing)
426         sendToAllClients("WARNING - Server has stopped accepting clients.");
427 }
428
429 /**
430  * This method overrides the one in the superclass. Called
431  * when the server closes down.
432  */
433 protected void serverClosed() {
434     serverUI.display("Server is closed.");
435 }
436
437 /**
438  * This method overrides the one in the superclass. Called
439  * when the server stops listening for connections.
440  */
441 protected void listeningException(Throwable exception) {
442     serverUI.display("An error has occurred while listening.");
443 }
444
445 /**
446  * This method is called when a client connects to the server.
447  * Added in phase 2.
448  * @param client The connection to the client who just connected.
449  */
450 protected void clientConnected(ConnectionToClient client) {
451     serverUI.display("A new client is attempting to connect to the server.");
452     client.setInfo("loginID", "");
453     client.setInfo("channel", "");
454     client.setInfo("passwordVerified", new Boolean(false));
455     client.setInfo("creatingNewAccount", new Boolean(false));
456     client.setInfo("fwdClient", "");
457     client.setInfo("blockedUsers", new Vector());
458
459     try {
460         client.sendToClient("Enter your login ID:");
461     } catch (IOException e) {
462         try {
463             client.close();
464         } catch (IOException ex) {}
465     }
466 }
467
468 /**
469  * This method is called when a client disconnects from the server.
```



```
470     * Added in phase 2.
471     *
472     * @param client The connection to the client who disconnected.
473     */
474     protected synchronized void clientDisconnected(ConnectionToClient client) {
475         handleDisconnect(client);
476     }
477
478     /**
479     * This method is called when an exception is detected in
480     * ConnectionToClient.
481     *
482     * @param client The client who caused the exception
483     * @param exception The exception thrown.
484     */
485     synchronized protected void clientException(ConnectionToClient client, Throwable
exception) {
486         handleDisconnect(client);
487     }
488
489     // Private methods -----
490
491     private void handleCmdWhoiblock(ConnectionToClient client) {
492         Vector blocked;
493
494         // If the client is not the server
495         if (client != null) {
496             blocked = new Vector((Vector)(client.getInfo("blockedUsers")));
497         } else {
498             blocked = new Vector(blockedUsers);
499         }
500
501         Iterator blockedIterator = blocked.iterator();
502
503         // If some clients are blocked
504         if (blockedIterator.hasNext()) {
505             sendToClientOrServer(client, "BLOCKED USERS:");
506
507             // Send the list of blocked users to the client
508             while (blockedIterator.hasNext()) {
509                 String blockedUser = (String)blockedIterator.next();
510                 sendToClientOrServer(client, "Messages from " + blockedUser + " are
blocked.");
511             }
512         } else {
513             // No clients are blocked
514             sendToClientOrServer(client, "No blocking is in effect.");
515         }
516     }
517
518     private void handleCmdUnblock(String command, ConnectionToClient client) {
519         Vector blocked = null;
520         boolean removedUser = false;
521         String userToUnblock = null;
522
523         //If the client is not the server
524         if (client != null) {
525             blocked = (Vector)(client.getInfo("blockedUsers"));
526         } else {
527             blocked = blockedUsers;
```

```
528     }
529
530     // Check if any users were blocked.
531     // If none were, notify the client
532     if (blocked.size() == 0) {
533         sendToClientOrServer(client, "No blocking is in effect.");
534         return;
535     }
536
537     // Obtain the user to unblock. If no user is specified, then
538     // an exception will be thrown and all users will be removed.
539     try {
540         userToUnblock = command.substring(9);
541     } catch (StringIndexOutOfBoundsException e) {
542         // We will unblock all users
543         userToUnblock = "";
544     }
545
546     // If we want to unblock the server.
547     if (userToUnblock.toLowerCase().equals("server"))
548         userToUnblock = "server";
549
550     // Get rid of the blocked user or all blocked users
551     Iterator blockedIterator = blocked.iterator();
552     while (blockedIterator.hasNext()) {
553         String blockedUser = (String)blockedIterator.next();
554
555         if(blockedUser.equals(userToUnblock) || userToUnblock.equals("")) {
556             blockedIterator.remove();
557             removedUser = true;
558             sendToClientOrServer(client, "Messages from " + blockedUser + " will now be
displayed.");
559         }
560     }
561
562     // Display error if user not found
563     if(!removedUser) {
564         sendToClientOrServer(client, "Messages from " + userToUnblock + " were not
blocked.");
565     }
566 }
567
568 private void handleCmdBlock(String command, ConnectionToClient client) {
569     Vector addBlock = null;
570
571     // This next line will verify a client was specified. If not,
572     // return an error message.
573     try {
574         // If there is no specified user to block we will go
575         // to the catch block
576         String userToBlock = command.substring(7);
577
578         //If the user wants to block the server
579         if (userToBlock.toLowerCase().equals("server")) {
580             userToBlock = "server";
581         }
582
583         // If the user tries to block himself
584         if (userToBlock.equals(client.getInfo("loginID"))) {
585             try {
```

```
586         client.sendToClient("Cannot block the sending of messages to yourself.");
587     } catch(IOException ex) {
588         serverUI.display("Warning: Error sending message.");
589     }
590     return;
591 } else {
592     // Blocking another user
593     // Verify if the login to block is valid
594     if (isLoginUsed(userToBlock) || userToBlock.equals("server")) {
595         // If the user we want to block is online
596         if (isLoginBeingUsed(userToBlock, false) && !userToBlock.equals("server"))
597     {
598         ConnectionToClient toBlock = getClient(userToBlock);
599
600         // If that user is forwarding to the client requesting
601         // the block, end the forwarding and notify them both.
602         if (((String)(toBlock.getInfo("fwdClient"))).equals(((String)
603 (client.getInfo("loginID"))))) {
604             toBlock.setInfo("fwdClient", "");
605             try {
606                 toBlock.sendToClient("Forwarding to " + client.getInfo("loginID")
607                     + " has been cancelled because " + client.getInfo("loginID")
608                     + " is now blocking messages from you.");
609
610                 client.sendToClient("Forwarding from " + toBlock.getInfo("loginID")
611                     + " to you has been terminated.");
612             } catch(IOException ioe) {
613                 serverUI.display("Warning: Error sending message.");
614             }
615         }
616
617         //Add the blocked user to the user's blocked users vector
618         addBlock = (Vector)(client.getInfo("blockedUsers"));
619         addBlock.addElement(userToBlock);
620     }
621     //If the user is trying to block a non-existing user.
622     else {
623         try {
624             client.sendToClient("User " + userToBlock + " does not exist.");
625         } catch(IOException ioe) {
626             serverUI.display("Warning: Error sending message.");
627         }
628         return;
629     }
630
631     //Send confirmation to the client that the user's messages will now be
632     blocked.
633     try {
634         client.sendToClient("Messages from " + userToBlock + " will be blocked.");
635     } catch(IOException ex) {
636         serverUI.display("Warning: Error sending message.");
637     }
638 } catch(StringIndexOutOfBoundsException e) {
639     try {
640         client.sendToClient("ERROR - usage #block <loginID>");
641     } catch(IOException ex) {
642         serverUI.display("Warning: Error sending message.");
643     }
644 }
```

```
643     }
644 }
645
646 private void handleCmdFwd(String command, ConnectionToClient client) {
647     try {
648         String destineeName = command.substring(5);
649
650         try {
651             // If the client is trying to forward to himself.
652             if (destineeName.equals(client.getInfo("loginID"))) {
653                 client.sendToClient("ERROR - Can't forward to self");
654                 return;
655             } else {
656                 // If the client is trying to forward to the server
657                 if (destineeName.toLowerCase().equals("server")) {
658                     client.sendToClient("ERROR - Can't forward to SERVER");
659                     return;
660                 } else {
661                     // If the client specified a non-existing client.
662                     if (getClient(destineeName) == null) {
663                         client.sendToClient("ERROR - Client does not exist");
664                         return;
665                     }
666                 }
667             }
668         } catch (IOException e) {
669             serverUI.display("Warning: Error sending message.");
670         }
671
672         // Find out if we are already forwarding. This will be used
673         // later when we check for a forwarding loop
674         String tempFwdClient = (String)(client.getInfo("fwdClient"));
675
676         // Get the connection to the intended destinee.
677         ConnectionToClient destinee = getClient(destineeName);
678
679         // If the destinee is not blocking messages from the client
680         // requesting the forwarding.
681         if (!(((Vector)(destinee.getInfo("blockedUsers"))).contains((String)
682 (client.getInfo("loginID"))))) {
683             client.setInfo("fwdClient", destineeName);
684         } else {
685             try {
686                 client.sendToClient("Cannot forward to " + destineeName
687                     + " because " + destineeName + " is blocking messages from you.");
688             } catch (IOException e) {
689                 serverUI.display("Warning: Error sending message.");
690             }
691             return;
692         }
693
694         try {
695             // If the client can be forwarded to without causing a loop
696             if (isValidFwdClient(client)) {
697                 client.sendToClient("Messages will be forwarded to: " +
698 client.getInfo("fwdClient"));
699             } else {
700                 // Reset forwarding to original value
701                 client.setInfo("fwdClient", tempFwdClient);
702                 client.sendToClient("ERROR - Can't forward because a loop would result");
703             }
704         }
705     }
706 }
```

```
701     }
702     } catch(IOException e) {
703         serverUI.display("Warning: Error sending message.");
704     }
705 } catch (StringIndexOutOfBoundsException e) {
706     try {
707         client.sendToClient("ERROR - usage: #fwd <loginID>");
708     } catch(IOException ex) {
709         serverUI.display("Warning: Error sending message.");
710     }
711 }
712 }
713
714 private void handleCmdPub(String command, ConnectionToClient client) {
715     String sender = "";
716     try {
717         sender = (String)(client.getInfo("loginID"));
718     } catch(NullPointerException e) {
719         sender = "server";
720     }
721
722     try {
723         Thread[] clients = server.getClientConnections();
724
725         for (int i = 0; i < clients.length; i++) {
726             ConnectionToClient c = (ConnectionToClient)(clients[i]);
727
728             // If the client selected by the iterator is not blocking messages from the
729             sender.
730             if (!(((Vector)(c.getInfo("blockedUsers"))).contains(sender))
731                 && ((Boolean)(c.getInfo("passwordVerified"))).booleanValue()) {
732                 c.sendToClient("PUBLIC MESSAGE from " + sender
733                     + "> " + command.substring(5));
734             }
735
736             // If the server is not blocking messages from the sender.
737             if (!blockedUsers.contains(sender)) {
738                 serverUI.display("PUBLIC MESSAGE from " + sender
739                     + "> " + command.substring(5));
740             }
741         } catch(IOException e) {
742             serverUI.display("Warning: Error sending message.");
743         }
744     }
745
746     private void handleCmdChannel(String command, ConnectionToClient client) {
747         String oldChannel = (String)(client.getInfo("channel"));
748
749         // Default new channel is the original channel that users start in
750         String newChannel = "main";
751
752         if(command.length() > 9)
753             newChannel = command.substring(9);
754
755         client.setInfo("channel", newChannel);
756         if (!oldChannel.equals("main")) {
757             sendChannelMessage(client.getInfo("loginID")
758                 + " has left channel: " + oldChannel, oldChannel, "");
759         }
```

```
760
761     if (!newChannel.equals("main")) {
762         sendChannelMessage(client.getInfo("loginID")
763             + " has joined channel: " + newChannel, newChannel, "");
764     }
765
766     // If the server receives all messages or is in the same channel
767     // as the client requesting the change, it will display a message
768     // indicating the change.
769     if (serverChannel == null || serverChannel.equals(client.getInfo("channel"))) {
770         serverUI.display(client.getInfo("loginID") + " has joined channel: " +
newChannel);
771     }
772 }
773
774 private void handleCmdPrivate(String command, ConnectionToClient client) {
775     try {
776         // Indicates where the spaces are in the command
777         int firstSpace = command.indexOf(" ");
778         int secondSpace = command.indexOf(" ", firstSpace + 1);
779
780         // Separate the different parts of the command
781         // These can throw the StringIndexOutOfBoundsException
782         String sender = "";
783         String loginID = command.substring(firstSpace + 1, secondSpace);
784         String message = command.substring(secondSpace + 1);
785
786         try {
787             sender = (String)(client.getInfo("loginID"));
788         } catch (NullPointerException e) {
789             sender = "server";
790         }
791
792         // If the message is for the server, display it and return
793         if (loginID.toLowerCase().equals("server")) {
794             //If the server is not blocking messages from the sender
795             if (!blockedUsers.contains(sender)) {
796                 serverUI.display("PRIVATE MESSAGE from " + sender + "> " + message);
797             }
798             //If the server is blocking messages from the sender.
799             else {
800                 try {
801                     client.sendToClient("Cannot send message because " + loginID + " is
blocking messages from you.");
802                 } catch (IOException e) {
803                     serverUI.display("Warning: Error sending message.");
804                 }
805             }
806         }
807         // If the message is not for the server
808         else {
809             try {
810                 Thread[] clients = server.getClientConnections();
811
812                 //Iterate through all the clients to find the destinee
813                 for (int i = 0; i < clients.length; i++) {
814                     ConnectionToClient c = (ConnectionToClient)(clients[i]);
815
816                     if (c.getInfo("loginID").equals(loginID)) {
```

```

817 // Once found, check if the user is not blocking messages from the
sender.
818 if (!(((Vector)(c.getInfo("blockedUsers"))).contains(sender))) {
819
820     // If he is not, check for a client to forward messages to.
821     if (!c.getInfo("fwdClient").equals("")) {
822         getFwdClient(c, sender).sendToClient("Forwarded> PRIVATE MESSAGE
from "
823             + sender + " to " + c.getInfo("loginID") + "> " + message);
824     } else {
825         c.sendToClient("PRIVATE MESSAGE from " + sender + "> " +
message);
826     }
827     serverUI.display("Private message: \"\"
828         + message + "\" from " + sender + " to " + c.getInfo("loginID"));
829 }
830 //If the user is blocking messages from the sender.
831 else {
832     sendToClientOrServer(client, "Cannot send message because "
833         + loginID + " is blocking messages from you.");
834 }
835 }
836 }
837 } catch(IOException e) {
838     serverUI.display("Warning: Error sending message.");
839 }
840 }
841 } catch (StringIndexOutOfBoundsException e) {
842     sendToClientOrServer(client, "ERROR - usage: #private <loginID> <msg>");
843 }
844 }
845
846 private void checkForBlocks(String login, ConnectionToClient client) {
847     String results = "User block check:";
848
849     if (!login.equals("server")) {
850         if (blockedUsers.contains(login))
851             results += "\nThe server is blocking messages from you.";
852     }
853
854     Thread[] clients = server.getClientConnections();
855
856     for (int i = 0; i < clients.length; i++) {
857         ConnectionToClient c = (ConnectionToClient)(clients[i]);
858
859         Vector blocked = (Vector)(c.getInfo("blockedUsers"));
860         if (blocked.contains(login)) {
861             results += "\nUser " + c.getInfo("loginID") + " is blocking your messages.";
862         }
863     }
864     if (results.equals("User block check:")) {
865         results += "\nNo user is blocking messages from you.";
866     }
867
868     sendToClientOrServer(client, results);
869 }
870
871 private boolean isValidFwdClient(ConnectionToClient client) {
872     boolean clientFound = false;
873     ConnectionToClient testClient = client;

```



```
874
875 // This block will make sure the client exists
876 Thread[] clients = server.getClientConnections();
877 for (int i = 0; i < clients.length; i++) {
878     ConnectionToClient tempc = (ConnectionToClient)(clients[i]);
879     if (tempc.getInfo("loginID").equals(testClient.getInfo("fwdClient"))) {
880         clientFound = true;
881     }
882 }
883
884 if (!clientFound)
885     return false;
886
887 // This block will check for endless loops
888 String theClients[] = new String[server.getNumberOfClients() + 1];
889 int i = 0;
890
891 // Loops until it finds a client that doesn't forward
892 while (testClient != null && testClient.getInfo("fwdClient")!="") {
893     // The name is added to the array
894     theClients[i] = (String)(testClient.getInfo("loginID"));
895
896     // If the name is in the array, return false as there is an endless loop
897     for(int j = 0; j < i; j++) {
898         if (theClients[j].equals(theClients[i]))
899             return false;
900     }
901     i++;
902
903     // Set "testClient" to the forwarded ConnectionToClient instance
904     testClient = getClient((String)testClient.getInfo("fwdClient"));
905 }
906 return true;
907 }
908
909 private ConnectionToClient getClient(String loginID) {
910     Thread[] clients = server.getClientConnections();
911
912     for (int i = 0; i < clients.length; i++) {
913         ConnectionToClient c = (ConnectionToClient)(clients[i]);
914         if (c.getInfo("loginID").equals(loginID))
915             return c;
916     }
917     return null; // If client wasn't found, return null
918 }
919
920 private void clientLoggingIn(String message, ConnectionToClient client) {
921     // Ignore blanks, if the user just hits 'enter'
922     if (message.equals(""))
923         return;
924
925     // If the client has not logged in yet and has entered
926     // guest as his login, create a new account
927     if ((client.getInfo("loginID").equals("")) && (message.equals("guest"))) {
928         // Save a flag so that when the next message arrives we
929         // know that it is the login ID for the new account
930         client.setInfo("creatingNewAccount", new Boolean(true));
931
932         try {
933             client.sendToClient("\n*** CREATING NEW ACCOUNT ***\nEnter new LoginID :");
```

```

934     } catch(IOException e) {
935         try {
936             client.close();
937         } catch (IOException ex) {}
938     }
939 } else {
940     // If creating a new account, and the user has just submitted his new login,
process it
941     if ((client.getInfo("loginID").equals(""))
942         && (((Boolean)(client.getInfo("creatingNewAccount"))).booleanValue())) {
943         client.setInfo("loginID", message);
944         try {
945             client.sendToClient("Enter new password :");
946         } catch(IOException e) {
947             try {
948                 client.close();
949             } catch (IOException ex) {}
950         }
951     } else {
952         // If the client is creating a new account and has just
953         // entered the password, then process it
954         if ((!client.getInfo("loginID").equals(""))
955             && (((Boolean)(client.getInfo("creatingNewAccount"))).booleanValue())) {
956             // If the login is not in the password file, accept the new account
957             if (!isLoginUsed(((String)(client.getInfo("loginID"))))) {
958                 client.setInfo("passwordVerified", new Boolean(true));
959                 client.setInfo("creatingNewAccount", new Boolean(false));
960                 client.setInfo("channel", "main");
961
962                 addClientToRegistry(((String)(client.getInfo("loginID")), message);
963                 serverUI.display(client.getInfo("loginID") + " has logged on.");
964                 sendToAllClients(client.getInfo("loginID") + " has logged on.");
965             } else {
966                 // If creating a new account, but the login is already used then keep
prompting for a login
967                 client.setInfo("loginID", "");
968                 client.setInfo("creatingNewAccount", new Boolean(false));
969                 try {
970                     client.sendToClient("Login already in use. Enter login ID:");
971                 } catch(IOException e) {
972                     try {
973                         client.close();
974                     } catch (IOException ex) {}
975                 }
976             }
977         }
978         // If the client is not creating a new account and has entered a login
979         else {
980             if (client.getInfo("loginID").equals("")) {
981                 client.setInfo("loginID", message);
982                 try {
983                     client.sendToClient("Enter password:");
984                 } catch(IOException e) {
985                     try {
986                         client.close();
987                     } catch (IOException ex) {}
988                 }
989             } else {
990                 // If the client is not creating a new account and has entered a
password

```

```

991 // Verify the client's login.
992 if ((isValidPwd((String)(client.getInfo("loginID")), message, true))
993     && (!isLoginBeingUsed((String)(client.getInfo("loginID")), true))) {
994     client.setInfo("passwordVerified", new Boolean(true));
995     client.setInfo("channel", "main");
996
997     // notify all users that a new client has logged on
998     serverUI.display(client.getInfo("loginID") + " has logged on.");
999     sendToAllClients(client.getInfo("loginID") + " has logged on.");
1000 } else {
1001     // If the login id or the password is invalid
1002     try {
1003         if (isLoginBeingUsed((String)(client.getInfo("loginID")), true)) {
1004             client.setInfo("loginID", "");
1005             client.sendToClient("Login ID is already logged on.\nEnter
LoginID:");
1006         } else {
1007             client.setInfo("loginID", "");
1008             client.sendToClient("\nIncorrect login or password\nEnter
LoginID:");
1009         }
1010     } catch (IOException e) {
1011         try {
1012             client.close();
1013         } catch (IOException ex) {}
1014     }
1015 }
1016 }
1017 }
1018 }
1019 }
1020 }
1021
1022 private void addClientToRegistry(String clientLoginID, String clientPassword) {
1023     try {
1024         // Part 1 : Transfer the data from the password file to a character buffer
1025         FileInputStream inputFile = new FileInputStream(PASSWORDFILE);
1026         byte buff[] = new byte[inputFile.available()];
1027
1028         for (int i = 0; i < buff.length; i++) {
1029             int character = inputFile.read();
1030             buff[i] = (byte)character;
1031         }
1032         inputFile.close(); // Close the input stream
1033
1034         // Part 2 : Delete the password file since it will be created again
1035         File fileToBeDeleted = new File(PASSWORDFILE);
1036         fileToBeDeleted.delete();
1037
1038         // Part 3 : Transfer the buffer and the client data to a new password file
1039         // with the same name as the first
1040         FileOutputStream outputFile = new FileOutputStream(PASSWORDFILE);
1041         for(int i = 0; i < buff.length; i++) // Write the buffer
1042             outputFile.write(buff[i]);
1043
1044         for(int i = 0; i < clientLoginID.length(); i++)
1045             outputFile.write(clientLoginID.charAt(i));
1046
1047         outputFile.write(SPACE); // Write a space character

```

```
1048     for (int i = 0; i < clientPassword.length(); i++)
1049         outputFile.write(clientPassword.charAt(i));
1050
1051     outputFile.write(RETURN); // Write a carriage return
1052     outputFile.write(LINEBREAK); // Write a line break
1053
1054     outputFile.close(); // Close the output stream
1055 } catch (IOException e) {
1056     serverUI.display("ERROR - Password File Not Found");
1057 }
1058 }
1059
1060 private boolean isLoginUsed(String loginID) {
1061     //See if the loginID is in the password file. The "false"
1062     //indicates not to verify the password
1063     return isValidPwd(loginID, "", false);
1064 }
1065
1066 private boolean isValidPwd(String loginID, String password, boolean
verifyPassword) {
1067     try {
1068         FileInputStream inputFile = new FileInputStream(PASSWORDFILE);
1069         boolean eoln = false; // Flag indicating the End Of Line
1070         boolean eof = false; // Flag indicating the End Of File
1071
1072         while (!eof) {
1073             eoln = false;
1074             String str = "";
1075             while (!eoln) {
1076                 int character = inputFile.read();
1077
1078                 if(character == -1) {
1079                     eof = true;
1080                     break;
1081                 } else {
1082                     if (character == LINEBREAK) {
1083                         eoln = true;
1084
1085                         // Verifies if the loginID is identical to the loginID
1086                         // in the file and, if necessary, verifies if the
1087                         // password is also identical to the password in the
1088                         // file
1089                         if ((str.substring(0, str.indexOf(" ")).equals(loginID))
1090                             && ((str.substring(str.indexOf(" ") + 1).equals(password)) ||
(!verifyPassword))) {
1091                             return true;
1092                         }
1093
1094                         // This condition checks if the char is anything other
1095                         // than a carriage return. The carriage return is
1096                         // ignored therefore there is no need to handle it
1097                     } else {
1098                         if (character != RETURN) {
1099                             str = str + (char)character;
1100                         }
1101                     }
1102                 }
1103             }
1104         }
1105         inputFile.close(); // Close the input stream
```

```
1106     } catch (IOException e) {
1107         serverUI.display("ERROR - Password File Not Found");
1108     }
1109     return false;
1110 }
1111
1112 private boolean isLoginBeingUsed(String loginID, boolean checkForDup) {
1113     boolean used = !checkForDup;
1114
1115     if (loginID.toLowerCase().equals("server"))
1116         return true;
1117
1118     // Creates an Iterator containing all the clients
1119     Thread[] clients = server.getClientConnections();
1120
1121     for (int i = 0; i < clients.length; i++) {
1122         ConnectionToClient tempc = (ConnectionToClient)(clients[i]);
1123         if (tempc.getInfo("loginID").equals(loginID)) {
1124             if (used)
1125                 return true;
1126
1127             used = true;
1128         }
1129     }
1130     return false; // The name was not found
1131 }
1132
1133 private void sendChannelMessage(String message, String channel, String login) {
1134     Thread[] clients = server.getClientConnections();
1135
1136     for (int i = 0; i < clients.length; i++) {
1137         ConnectionToClient c = (ConnectionToClient)(clients[i]);
1138
1139         if (c.getInfo("channel").equals(channel)
1140             && !(((Vector)(c.getInfo("blockedUsers"))).contains(login))) {
1141             try {
1142                 if (!(c.getInfo("fwdClient").equals(""))) {
1143                     getFwdClient(c, login).sendToClient("Forwarded> " + message);
1144                 } else {
1145                     c.sendToClient(message);
1146                 }
1147             } catch (IOException e) {
1148                 serverUI.display("Warning: Error sending message.");
1149             }
1150         }
1151     }
1152 }
1153
1154 private ConnectionToClient getFwdClient(ConnectionToClient c, String sender) {
1155     Vector pastRecipients = new Vector();
1156
1157     //Add the first recipient to the vector
1158     pastRecipients.addElement((String)(c.getInfo("loginID")));
1159
1160     // Loops until it finds a client that doesn't forward messages
1161     while (!c.getInfo("fwdClient").equals("")) {
1162         Thread[] clients = server.getClientConnections();
1163
1164         for (int i = 0; i < clients.length; i++) {
1165             ConnectionToClient tempc = (ConnectionToClient)(clients[i]);
```

```

1166         if (tempc.getInfo("loginID").equals(c.getInfo("fwdClient"))) {
1167             // We have found the client being forwarded to by c
1168             // Now check that c is not blocking the original sender
1169             if (!(((Vector)(tempc.getInfo("blockedUsers"))).contains(sender))) {
1170                 //Look in the previous recipients to see if any of them are blocked.
1171                 Iterator pastIterator = pastRecipients.iterator();
1172
1173                 while (pastIterator.hasNext()) {
1174                     String pastRecipient = (String)pastIterator.next();
1175                     if (((Vector)(tempc.getInfo("blockedUsers"))).contains(pastRecipient))
1176 {
1177                     //This means one of the past recipients is blocked
1178                     //by the client supposed to be forwarded to.
1179                     try {
1179                         c.sendToClient("Cannot forward message. A past recipient of this
message is blocked by "
1180 + (String)(tempc.getInfo("loginID")));
1181                     } catch (IOException e) {
1182                         serverUI.display("Warning: Error sending message.");
1183                     }
1184                     return c;
1185                 }
1186             }
1187
1188             // Now continue looking for further forwarding if necessary
1189             if (!tempc.getInfo("fwdClient").equals("")) {
1190                 c = tempc;
1191                 pastRecipients.addElement((String)(c.getInfo("loginID")));
1192             } else {
1193                 return tempc;
1194             }
1195         } else {
1196             try {
1197                 c.sendToClient("Cannot forward message. Original sender is blocked by
"
1198 + ((String)(c.getInfo("fwdClient"))));
1199             } catch (IOException e) {
1200                 serverUI.display("Warning: Error sending message.");
1201             }
1202             return c;
1203         }
1204     }
1205 }
1206 }
1207 return c;
1208 }
1209
1210 private void sendListOfClients(ConnectionToClient c) {
1211     Vector clientInfo = new Vector();
1212
1213     Thread[] clients = server.getClientConnections();
1214
1215     for (int i = 0; i < clients.length; i++) {
1216         ConnectionToClient tempc = (ConnectionToClient)(clients[i]);
1217         clientInfo.addElement((String)(tempc.getInfo("loginID"))
1218 + " --- on channel: " + (String)(tempc.getInfo("channel")));
1219     }
1220
1221     //Sort the vector containing the information.
1222     Collections.sort(clientInfo);

```

```
1223
1224     if (server.isListening() || server.getNumberOfClients() != 0) {
1225         sendToClientOrServer(c, "SERVER --- on channel: "
1226             + (serverChannel == null ? "main" : serverChannel));
1227     } else {
1228         serverUI.display("SERVER --- no active channels");
1229     }
1230
1231     Iterator toReturn = clientInfo.iterator();
1232
1233     while (toReturn.hasNext()) {
1234         sendToClientOrServer(c, (String)toReturn.next());
1235     }
1236 }
1237
1238 private void handleServerCmdBlock(String message) {
1239     try {
1240         String userToBlock = message.substring(7);
1241
1242         if (userToBlock.toLowerCase().equals("server")) {
1243             serverUI.display("Cannot block the sending of messages to yourself.");
1244             return;
1245         } else {
1246             if (isLoginUsed(userToBlock)) {
1247                 blockedUsers.addElement(userToBlock);
1248             } else {
1249                 serverUI.display("User " + userToBlock + " does not exist.");
1250                 return;
1251             }
1252         }
1253
1254         serverUI.display("Messages from " + userToBlock + " will be blocked.");
1255     } catch (StringIndexOutOfBoundsException e) {
1256         serverUI.display("ERROR - usage #block <loginID>");
1257     }
1258 }
1259
1260 private void handleServerCmdPunt(String message) {
1261     Thread[] clients = server.getClientConnections();
1262
1263     try {
1264         //Iterate to get the connection to the client we want to expell
1265         for (int i = 0; i < clients.length; i++) {
1266             ConnectionToClient c = (ConnectionToClient)(clients[i]);
1267             if (c.getInfo("loginID").equals(message.substring(6))) {
1268                 //Ignore the exception that might occur as we only want
1269                 //to get rid of this user.
1270                 try {
1271                     c.sendToClient("You have been expelled from this server.");
1272                 } catch (IOException e) {}
1273                 finally {
1274                     try {
1275                         c.close();
1276                     }
1277                     catch (IOException ex) {}
1278                 }
1279             }
1280         }
1281     } catch (StringIndexOutOfBoundsException ex) {
1282         serverUI.display("Invalid use of the #punt command.");
1283     }
```



```
1283     }
1284 }
1285
1286 private void handleServerCmdWarn(String message) {
1287     Thread[] clients = server.getClientConnections();
1288
1289     try {
1290         for (int i = 0; i < clients.length; i++) {
1291             ConnectionToClient c = (ConnectionToClient)(clients[i]);
1292             if (c.getInfo("loginID").equals(message.substring(6))) {
1293                 //If an exception occurs, boot the user being warned.
1294                 //He is causing more trouble than he's worth!
1295                 try {
1296                     c.sendToClient("Continue and you WILL be expelled.");
1297                 } catch (IOException e) {
1298                     try {
1299                         c.close();
1300                     } catch (IOException ex) {}
1301                 }
1302             }
1303         }
1304     } catch (StringIndexOutOfBoundsException ex) {
1305         serverUI.display("Invalid use of the #warn command.");
1306     }
1307 }
1308
1309 private void sendToClientOrServer(ConnectionToClient client, String message) {
1310     try {
1311         client.sendToClient(message);
1312     } catch (NullPointerException npe) {
1313         serverUI.display(message);
1314     } catch (IOException ex) {
1315         serverUI.display("Warning: Error sending message.");
1316     }
1317 }
1318
1319 private void handleDisconnect(ConnectionToClient client) {
1320     if (!client.getInfo("loginID").equals("")) {
1321         try {
1322             Thread[] clients = server.getClientConnections();
1323
1324             // Remove any forwarding to this client by others.
1325             for (int i = 0; i < clients.length; i++) {
1326                 ConnectionToClient c = (ConnectionToClient)(clients[i]);
1327                 if (client.getInfo("loginID").equals(c.getInfo("fwdClient"))) {
1328                     c.setInfo("fwdClient", "");
1329                     c.sendToClient("Forwarding to " + client.getInfo("loginID")+ " has been
cancelled.");
1330                 }
1331             }
1332             sendToAllClients(((client.getInfo("loginID") == null) ?
"" : client.getInfo("loginID")) + " has disconnected.");
1333         } catch (IOException e) {
1334             serverUI.display("Warning: Error sending message.");
1335         }
1336         serverUI.display(client.getInfo("loginID") + " has disconnected.");
1337     }
1338 }
1339 }
1340 }
```