Distributed Systems

Exercise 9

Richard Möhn (201311231) Mathias Dannesbo (201206106)

February 9, 2014

1 Introduction

We used pairprogramming for all the code and "pairreporting" for the report, so we share the workload at 50% each.

In this report we present an editor the instances of which are able to connect to each other and to capture editing actions in one editor and replay them in another. For this we were provided with an editor that already captured editing events in one text pane and replayed them in another. Our contribution was to enable capture and replay between two editors over a TCP/IP connection.

The report describes how we filled previously meaningless menu items with life, so that users can set up and tear down connections with them. It also shows our design for transmitting events between editors and for the process of disconnecting two connected editors cleanly. It discusses some of the decisions we made in developing the editor. The conclusion contains a list of issues with the editor that still need to be addressed.

2 Code Overview

We need to preform actions when the `Listen`, `Connect` and `Disconnect` items are clicked in the menus. The items represent `javax.swing.Action`s whose `actionPerformed()` methods we filled with code.

2.1 Listen

The `Listen` is parsing the server information from the GUI and putting a new Runnable in the AWT system `EventQueue` which start up a new `ServerSocket` for listening for another editor to connect as a client. Then it calls `startCommunication`, which is described in section 2.3. Finally it sets the title of the editor.

File: DistributedTextEditor.java

```
139     Action Listen = new AbstractAction("Listen") {
140         private static final long serialVersionUID = 3098L;
141 
142         public void actionPerformed(ActionEvent e) {
143             saveOld();
144 
145             // Prepare for connection
146             area1.setText("");
147             changed = false;
148             Save.setEnabled(false);
149             SaveAs.setEnabled(false);
150 
151             // Display information about the listening
152             String address = null;
153             try {
154                 address = InetAddress.getLocalHost().getHostAddress();
155             }
156             catch (UnknownHostException ex) {
157                 ex.printStackTrace();
158                 System.exit(1);
159             }
160             final int port = Integer.parseInt(portNumber.getText());
161             setTitle(
162                 String.format("I'm listening on %s:%d.", address, port));
163 
164             // "Asynchronously" wait for a connection
165             EventQueue.invokeLater( new Runnable() {
166                 public void run() {
167                     // Wait for an incoming connection
168                     Socket socket = null;
169                     try {
170                         ServerSocket servSock = new ServerSocket(port);
171                         socket = servSock.accept();
172                         servSock.close();
173                     }
174                     catch (IOException ex) {
175                         ex.printStackTrace();
176                         System.exit(1);
177                     }
178 
179                     // Set up the event sending and receiving
```

```
180                     startCommunication(socket, inEventQueue, outEventQueue);
181
182                     // Give the editor a better title
183                     setTitle(
184                         String.format(
185                             "Connected to %s:%d.",
186                             socket.getInetAddress().toString(),
187                             socket.getPort()
188                         )
189                     );
190                 }
191             } );
192         }
193     };
```

2.2 Connect

The `Connect` first clears the textareas, then parses the server information from the GUI and connect to the corresponding socket. As soon as the TCP/IP connection is etablished, it calls `startCommunication` which is described in section 2.3. Finally it set the title of the editor.

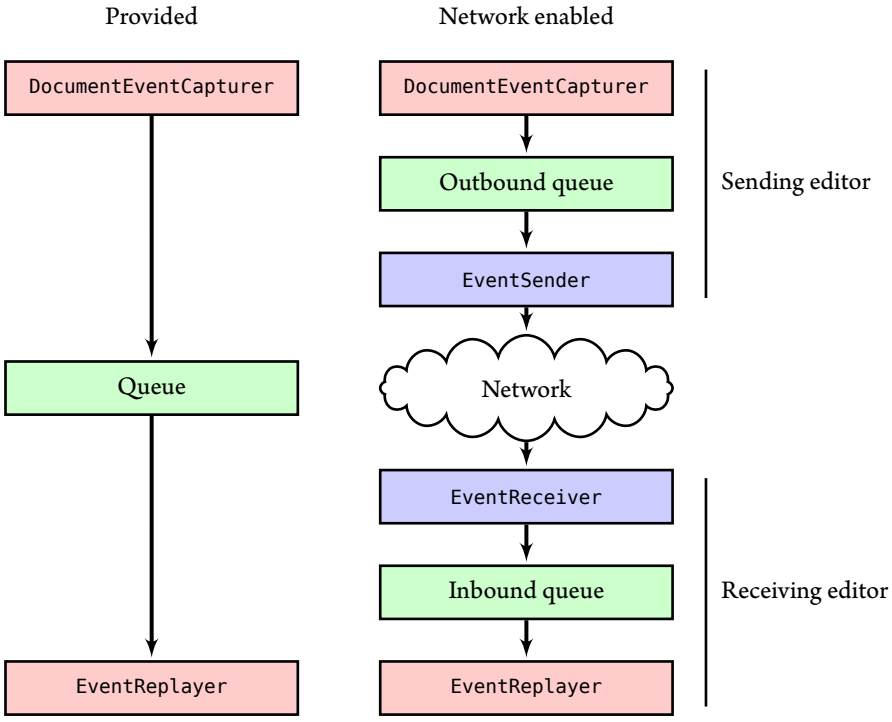
File: DistributedTextEditor.java

```
195     Action Connect = new AbstractAction("Connect") {
196         private static final long serialVersionUID = 135098L;
197
198         public void actionPerformed(ActionEvent e) {
199             // Prepare for connection
200             saveOld();
201             area1.setText("");
202             changed = false;
203             Save.setEnabled(false);
204             SaveAs.setEnabled(false);
205
206             // Find out with whom to connect
207             String address = ipaddress.getText();
208             int port = Integer.parseInt( portNumber.getText() );
209             setTitle(
210                 String.format("Connecting to %s:%d...", address, port));
211
212             // Initiate connection with other editor
213             Socket socket = null;
```

```
214             try {
215                 socket = new Socket(address, port);
216             } catch (IOException ex) {
217                 ex.printStackTrace();
218                 System.exit(1);
219             }
220 
221             // Set up the event sending and receiving
222             startCommunication(socket, inEventQueue, outEventQueue);
223 
224             // Give the editor a better title
225             setTitle(
226                 String.format("Connected to %s:%d.", address, port));
227         }
228     };
```

2.3 Communication Between Editors

Figure 1: `MyTextEvent`'s path through the system in the editor as provided and in our version.



In the not-networked editor, the `DocumentEventCapturer` and the `EventReplayer` worked together directly and communicated over one queue: When the user wrote in the upper pane, the `DocumentEventCapturer` recorded the edit events and put them in the queue. Afterwards the `EventReplayer` retrieved them from the queue and applied them to the lower pane. See the left part of figure 1.

Now we have two editors (A and B) with one lower and one upper pane on each of them. Edit events from the upper pane of A have to be transmitted to the lower pane of B and vice versa. We therefore need two queues: One for the transmission from A to B and one for the transmission from B to A.

However, as the editors are different processes on possibly different machines, the queues have to be split up: Each has a head on one of the peers and a tail on the other. Not using RMI, `EventSender` and `EventReceiver` establish the connection between the queues over the network manually. Overall it works like this: `DocumentEventCapturer` puts events in a queue as before and `EventReplayer` retrieves events from a queue as before. But now, those are different queues. `EventSender` retrieves the elements `DocumentEventCapturer` put in an outbound queue and sends them over the network. `EventReceiver` receives events from the network and puts them in an inbound queue off which `EventReplayer` feeds. This establishes a persistent and asynchronous way of communication.

Since the network operations `writeObject` and `readObject` can play the role of blocking communication, queues are not absolutely necessary. Instead we could just have modified the `DocumentEventCapturer` so that it sends events over the network instead of putting them in a queue and we could have modified the `EventReplayer` so that it receives events from the network instead of taking them out of a queue. But adding a layer (layered architecture) between those classes and the network had the advantage that we didn't have to change them very much. On top of that, it would have been a bad design: Classes should only be responsible for one thing at a time. The message-queueing communication style also prevents the UI thread from being blocked or slowed down by network communication and thereby increases distribution transparency.

Continuing to use queues enable us to leave `EventReplayer` and `DocumentEventCapturer` largely unchanged. We just had to take the queue out of the latter, and make the former take events directly from a queue instead of indirectly through the `DocumentEventCapturer`. The following listings show the results of these changes.

File: EventReplayer.java

```
1 package ddist;
2
3 import java.awt.EventQueue;
4 import java.util.concurrent.BlockingQueue;
5
6 import javax.swing.JFrame;
7 import javax.swing.JOptionPane;
```

```
8  import javax.swing.JTextArea;
9
10 /**
11  *
12  * Takes the event recorded by the DocumentEventCapturer and replays
13  * them in a JTextArea.
14  *
15  * @author Jesper Buus Nielsen
16  *
17  */
18 public class EventReplayer implements Runnable {
19
20     private BlockingQueue<MyTextEvent> eventQueue;
21     private JTextArea area;
22     private JFrame frame;
23
24     /**
25      * @param eventQueue the blocking queue from which to take events to
26      * replay them in the on the second argument
27      * @param area the text area in which to replay the events
28      * @param frame the overall frame of the program (might be done better)
29      */
30     public EventReplayer(BlockingQueue<MyTextEvent> eventQueue,
31                          JTextArea area, JFrame frame) {
32         this.eventQueue = eventQueue;
33         this.area = area;
34         this.frame = frame;
35     }
36
37     public void run() {
38         boolean wasInterrupted = false;
39         while (!wasInterrupted) {
40             try {
41                 MyTextEvent mte = eventQueue.take();
42                 if (mte instanceof TextInsertEvent) {
43                     final TextInsertEvent tie = (TextInsertEvent)mte;
44                     EventQueue.invokeLater(new Runnable() {
45                             public void run() {
46                                 try {
47                                     area.insert(tie.getText(), tie.getOffset());
48                                 } catch (Exception e) {
```

```
49                                  System.err.println(e);
50                                  /* We catch all axceptions, as an uncaught
51                                   * exception would make the EDT unwind,
52                                   * which is now healthy.
53                                   */
54                              }
55                          }
56                      });
57              } else if (mte instanceof TextRemoveEvent) {
58                  final TextRemoveEvent tre = (TextRemoveEvent)mte;
59                  EventQueue.invokeLater(new Runnable() {
60                          public void run() {
61                              try {
62                                  area.replaceRange(null, tre.getOffset(),
63                                                    tre.getOffset() +
64                                                    tre.getLength());
65                              } catch (Exception e) {
66                                  System.err.println(e);
67                                  /* We catch all axceptions, as an uncaught
68                                   * exception would make the EDT unwind,
69                                   * which is now healthy.
70                                   */
71                              }
72                          }
73                      });
74              }
75              else if (mte instanceof DisconnectEvent) {
76                  JOptionPane.showMessageDialog(frame, "Disconnected.");
77                  frame.setTitle("Disconnected");
78                  area.setText("");
79              }
80              else {
81                  System.err.println("Illegal event received.");
82                  System.exit(1);
83              }
84          } catch (Exception _) {
85              wasInterrupted = true;
86          }
87      }
88      System.out.println(
89          "I'm the thread running the EventReplayer, now I die!");
```

```
90      }
91  }
```

File: DistributedTextEditor.java

```
43      /*
44       * Queue for holding events coming in from the other editor to be written
45       * to the lower text area.
46       */
47      private BlockingQueue<MyTextEvent> inEventQueue
48          = new LinkedBlockingQueue<>();
49
50      /*
51       * Queue for holding events coming from the upper text area to be sent to
52       * the other editor.
53       */
54      private BlockingQueue<MyTextEvent> outEventQueue
55          = new LinkedBlockingQueue<>();
56
57      private EventReplayer er;
58      private Thread ert;
59
60      private JFileChooser dialog =
61              new JFileChooser(System.getProperty("user.dir"));
62
63      private String currentFile = "Untitled";
64      private boolean changed = false;
65      private boolean connected = false;
66      private DocumentEventCapturer dec
67          = new DocumentEventCapturer(outEventQueue);
```

File: DocumentEventCapturer.java

```
30      protected BlockingQueue<MyTextEvent> eventHistory;
31
32      /**
33       * @param eventHistory the queue this object should write the captured
34       * events to
35       */
36      public DocumentEventCapturer(BlockingQueue<MyTextEvent> eventHistory) {
37          this.eventHistory = eventHistory;
38      }
39
```

To set up the communication threads, the `Listen` and `Connect` actions call the method `startCommunication` of the `DistributedTextEditor` shown below. It starts the `EventSender` and `EventReceiver` as new threads since they have to send and receive events asynchronously.

File: DistributedTextEditor.java

```
300     /**
301      * Start threads for handling the transportation of events between the
302      * network and the local event queues.
303      */
304     private void startCommunication(Socket socket,
305             BlockingQueue<MyTextEvent> inEventQueue,
306             BlockingQueue<MyTextEvent> outEventQueue) {
307             // Start thread for adding incoming events to the inqueue
308             EventReceiver rec
309                 = new EventReceiver(socket, inEventQueue, outEventQueue);
310             Thread receiverThread = new Thread(rec);
311             receiverThread.start();
312 
313             // Start thread for taking outgoing events from the outqueue
314             EventSender sender
315                 = new EventSender(socket, inEventQueue, outEventQueue);
316             Thread senderThread = new Thread(sender);
317             senderThread.start();
318     }
```

The following listings show the `EventSender` and `EventReceiver`, which connect queues over the network as described above.

File: EventSender.java

```
1  package ddist;
2  
3  import java.io.IOException;
4  import java.io.ObjectOutputStream;
5  import java.net.Socket;
6  import java.util.concurrent.BlockingQueue;
7  
8  /**
9   * Thread responsible for retrieving MyTextEvents from an event queue and
10  * sending them to another editor.
11  */
12 public class EventSender implements Runnable {
13     private BlockingQueue<MyTextEvent> _inEventQueue;
```

```
14      private BlockingQueue<MyTextEvent> _outEventQueue;
15      private Socket _socket;
16
17      /**
18       * @param sock a Socket representing the connection to the other editor
19       * @param inEventQueue a BlockingQueue in which to place edit events from
20       * the other editor
21       * @param outEventQueue a BlockingQueue from which to take events for
22       * sending to the other editor
23       */
24      public EventSender(Socket sock, BlockingQueue<MyTextEvent> inEventQueue,
25              BlockingQueue<MyTextEvent> outEventQueue) {
26          _socket        = sock;
27          _inEventQueue  = inEventQueue;
28          _outEventQueue = outEventQueue;
29      }
30
31      public void run() {
32          try {
33              // Open connection to the other editor
34              ObjectOutputStream objOut
35              = new ObjectOutputStream( _socket.getOutputStream() );
36
37              // Send events arriving in the queue to other editor
38              while (true) {
39                  MyTextEvent event = _outEventQueue.take();
40                  objOut.writeObject(event);
41
42                  // Cleanup and close thread if we want to disconnect
43                  if (event instanceof DisconnectEvent) {
44                      // Do more cleanup if receiver thread already dead
45                      if ( ((DisconnectEvent) event).shouldClose() ) {
46                          _inEventQueue.clear();
47                          _outEventQueue.clear();
48                          _socket.close();
49                      }
50
51                      break;
52                  }
53              }
54          }
```

```
55          catch (IOException | InterruptedException e) {
56              e.printStackTrace();
57              System.exit(1);
58          }
59      }
60  }
```

File: EventReceiver.java

```
1   package ddist;
2
3   import java.io.IOException;
4   import java.io.ObjectInputStream;
5   import java.net.Socket;
6   import java.util.concurrent.BlockingQueue;
7
8   /**
9    * Thread responsible for receiving MyTextEvents and adding them to the event
10   * queue.
11   */
12  public class EventReceiver implements Runnable {
13      Socket _socket;
14      BlockingQueue<MyTextEvent> _inEventQueue;
15      BlockingQueue<MyTextEvent> _outEventQueue;
16
17      /**
18       * @param sock a Socket representing the connection to the other editor
19       * @param inEventQueue a BlockingQueue in which to place edit events from
20       * the other editor
21       * @param outEventQueue a BlockingQueue from which to take events for
22       * sending to the other editor
23       */
24      public EventReceiver(Socket sock, BlockingQueue<MyTextEvent> inEventQueue,
25              BlockingQueue<MyTextEvent> outEventQueue) {
26          _socket        = sock;
27          _inEventQueue  = inEventQueue;
28          _outEventQueue = outEventQueue;
29      }
30
31      public void run() {
32          try {
33              // Open connection to other editor
```

```
34              ObjectInputStream objIn
35                  = new ObjectInputStream( _socket.getInputStream() );
36
37              // Put edit events from the other editor in the queue
38              while (true) {
39                  MyTextEvent event = (MyTextEvent) objIn.readObject();
40                  _inEventQueue.put(event);
41
42                  // Cleanup and close thread if client wants to disconnect
43                  if (event instanceof DisconnectEvent) {
44                      DisconnectEvent disconnectEvent = (DisconnectEvent) event;
45
46                      // Do more cleanup if the other thread is dead already
47                      if ( disconnectEvent.shouldClose() ) {
48                          _inEventQueue.clear();
49                          _outEventQueue.clear();
50                          _socket.close();
51                      }
52                      // Otherwise say that the others should do more cleanup
53                      else {
54                          disconnectEvent.setShouldClose();
55                          _outEventQueue.put(event);
56                      }
57
58                      break;
59                  }
60              }
61          }
62          catch (IOException | InterruptedException | ClassNotFoundException e) {
63              e.printStackTrace();
64              System.exit(1);
65          }
66      }
67  }
```

2.4 Disconnect

`Disconnect` is a menu item like `Listen` and `Connect` and after it is clicked, the corresponding `actionPerformed` method printed below resets the editor to a disconnected state and kicks off the teardown of the connection.

File: DistributedTextEditor.java

```
230     Action Disconnect = new AbstractAction("Disconnect") {
231         private static final long serialVersionUID = 983498L;
232 
233         public void actionPerformed(ActionEvent e) {
234             setTitle("Disconnected");
235             area2.setText("");
236 
237             // Initiate disconnecting process
238             outEventQueue.add( new DisconnectEvent() );
239         }
240     };
```

Surprisingly, connection teardown is much more difficult to implement than connection setup, because a number of threads have to be notified that they should end their life. Since the only way to reach all threads is the events we already use for communicating text edit actions, we introduce a special event, the `DisconnectEvent`. By putting it in the outbound queue, the `Disconnect` menu action triggers a rather complicated process of closing the connection. The following listing contains the code of the `DisconnectEvent` along with a description of the process of closing the connection.

File: DisconnectEvent.java

```
1  package ddist;
2  
3  /**
4   * Event indicating that someone wants to disconnect.
5   */
6  public class DisconnectEvent extends MyTextEvent {
7      /*
8       * There are six threads that have to be notified when one user wants to
9       * disconnect:
10      *
11      *  MI: the main thread of the initiator of the disconnect
12      *  MR: the main thread of the receiver of the disconnect
13      *  II: the thread responsible for incoming events at the initiator
14      *  IR: the thread responsible for incoming events at the receiver
15      *  OI: the thread responsible for outgoing events at the initiator
16      *  OR: the thread responsible for outgoing events at the receiver
17      *
18      * The process of disconnection uses the normal communication paths
19      * between the threads and the editors. It looks like this:
20      *
```

```
21      * MI -> OI ----------------------> IR -> MR
22      *                                   |
23      *                                   v
24      *       II <----------------------- OR
25      *
26      *  1. The main thread at the initiator creates a DisconnectEvent and puts
27      *     it in the queue for outgoing events (outqueue). It also takes the
28      *     necessary actions to put the editor in a disconnected state.
29      *
30      *  2. The thread OI, responsible for taking elements from the outqueue and
31      *     sending them to the other editor, takes the DisconnectEvent from
32      *     the outqueue, sends it to the other editor and shuts down, because
33      *     of the special event.
34      *
35      *  3. The thread IR, responsible for receiving events from the other
36      *     editor and putting them in the queue for incoming events (inqueue),
37      *     receives the event and puts it in the inqueue. After this, the
38      *     socket is only used by one of the threads in each editor. Those
39      *     threads have to close it. IR indicates this to the threads coming
40      *     after it by setting the _shouldClose flag in the DisconnectEvent.
41      *     It puts the DisconnectEvent in the local outqueue and it shuts down.
42      *
43      *  4. OR takes the DisconnectEvent event from the outqueue and sends it
44      *     on. The DisconnectEvent now indicates that the socket should be
45      *     closed. OR complies and then shuts itself down.
46      *
47      *  5. II receives the DisconnectEvent event from the network, shuts the
48      *     socket down as told and terminates.
49      */
50
51     private static final long serialVersionUID = -3411878142976145233L;
52
53     // Indicates whether threads that see this event may close the socket
54     private boolean _shouldClose = false;
55
56     public DisconnectEvent() {
57         super(-1);
58     }
59
60     public void setShouldClose() {
61         _shouldClose = true;
```

```
62      }
63
64      public boolean shouldClose() {
65          return _shouldClose;
66      }
67  }
```

3 Conclusion

We have changed the provided rudimental text editor so that instances of it are capable of sending editing events over a network to each other and replaying them locally. The underlying means of communication is TCP, since it offers reliable and ordered transmission of data. Our application comprises two layers on top of the already layered TCP/IP stack and establishes persistent asynchronous communication by means of message queueing.

The hardest part in developing the system was to figure out how four threads and a networking connection should be torn down in a clean and ordered way. It is also here where some unexpected and yet unexplained behaviour still occurs.

There are some more points that might be improved: When an editor listens for connections, it does so in the main thread, so that the GUI freezes. The `DisconnectEvent` extending `MyTextEvent` is a case of implementation inheritance which isn't appropriate in this case. Instead, both should implement a common `Event` interface. Instead of handling exceptions, the editor terminates. Classes are tagged `Serializable` without caring for the implications of this.

All in all, however, we came up with a rather elegant design, which should make clearing up the above issues easy.

A Finding the Code and Running the Editor

The file `Code1864-ex09.zip` contains a Maven repository with the source code and a JAR file being the executable editor. From the root directory it can be run with `./run.sh`. The code can also be found online at `wiply.neic.dk/au/ddist/Code1864-ex09.zip`.