Distributed Systems

# Exercise 11

Richard Möhn (201311231) Mathias Dannesbo (201206106)

February 23, 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 *in the same text*. We built upon the editor from excercise 09, which was able to connect and send edit event to other instances of itself, but not in the same textarea. Our contribution was to make two editors merge their edits in the same text.

The report describes how we changed the architecture of the event flow based on operational transformation. 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 Operational transformation

There exist quite some applications for collaborative text editing; examples are Saros [4], Google Wave [2] or Gobby [1]. They all use a client-server approach for synchronising optimistically. This means, changes to the text are made immediately (when the user presses a key, for example) and then sent to the server, which in turn sends them to the other clients. If changes are made concurrently by multiple users, an algorithm makes sure that everyone ends up with the same text. This approach to synchronisation, or to preserving consistency, increases distribution transparency since there is no delay between user input and its effects on the screen, just as in a non-distributed editor.

One method for this kind of synchronisation is operational transformation: Operations represent changes to the text and they are first applied locally, then sent to the other parties editing the same text. If the other parties edit their versions of the text in the meantime, they have to transform the received operation so that it does what was intended.

Example: A and B start with text `abcdefg`. A inserts `123` at offset 2 and gets `ab123cdefg`, then sends the operation to B. Before B receives it, she removes two characters at offset 4 and gets `abcdg`, then she sends this operation to A. Now, A and B receive the operations from each other, but if they applied them directly, they would end up with different texts: A with `ab12defg` and B with `ab123cdg`. In this case, the result B gets already appears like a sensible combination of the two operations. Then only the remove operation A received has to be shifted by three characters to the right, so that it ends up with that text, too. Mathematically:

$$
\begin{aligned}
t &:= \texttt{abcdefg} \\
o_A &:= (\mathit{insert}, 2, \texttt{123}) \\
o_B &:= (\mathit{remove}, 4, 2) \\
o_A(t) &= \texttt{ab123cdefg} \\
o_B(t) &= \texttt{abcdg} \\
(o_B \circ o_A)(t) &= \texttt{ab12defg} \\
(o_A \circ o_B)(t) &= \texttt{ab123cdg} \\
(o'_A, o'_B) &:= \text{transform}(o_A, o_B) \\
o'_A &= (\mathit{insert}, 2, \texttt{123}) \\
o'_B &= (\mathit{remove}, 4, 5) \\
(o'_B \circ o_A)(t) &= (o'_A \circ o_B)(t) = \texttt{ab123cdg}
\end{aligned}
$$

The transform function is the first component of operational transformation. Its defining property is:

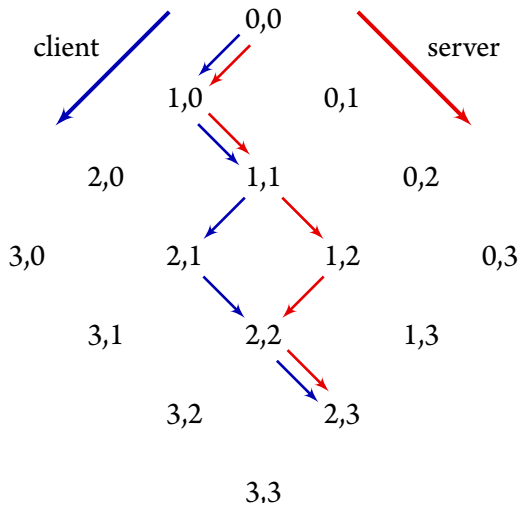
$$\forall o_a, o_b : (o'_a, o'_b) := \text{transform}(o_a, o_b) \Rightarrow (o'_b \circ o_a) = (o'_a \circ o_b)$$

That means, if two copies of the same text have diverged by applying $o_a$ to one and $o_b$ to the other, then applying $o'_b$ after $o_a$ and $o'_a$ after $o_b$ brings them in synch again. Any function that has the above property can be used as transform, but most of them will only give consistent, but not sensible, results.

The second component of operational transformation is an algorithm that wraps the transform function, sends operations to other parties, receives operations from other parties, transforms them and applies them locally. All the applications mentioned above use a version of the so-called Jupiter algorithm, which was originally published in [3]. By using a client-server approach it reduces the problem of synchronising $n$ parties to the problem of synchronising 2 parties.

[3] and [5] contain extensive descriptions of how it works, so we won't go into details here. However, figure 1 illustrates its workings. The numbers count the operations applied to the text on the client and server and, paired up, they are logical timestamps identifying the states the

Figure 1: The state space in the Jupiter algorithm.




edited text goes through. The arrows correspond to operations applied to the text, the state of which changes after every application of an operation. Operations include the timestamp of the state to which they were applied. Both the client and the server start in state 0,0. First, the client generates and applies an operation and the text goes into state 1,0. The client sends the operation to the server, which applies it, so that its copy of the text goes into state 1,0, too. The same happens the other way around when the server generates the second operation. Then the client applies an operation and the also server applies an operation, before receiving that of the client. The result are two different states of the text after the same total number of operations. Client and server now receive each other's operations, but cannot apply them because their texts are not in the state to which the operations where originally applied. In order to get from their diverged states into a common state again, client and server have to transform the received operations and apply the results. Then they proceed without conflicts.

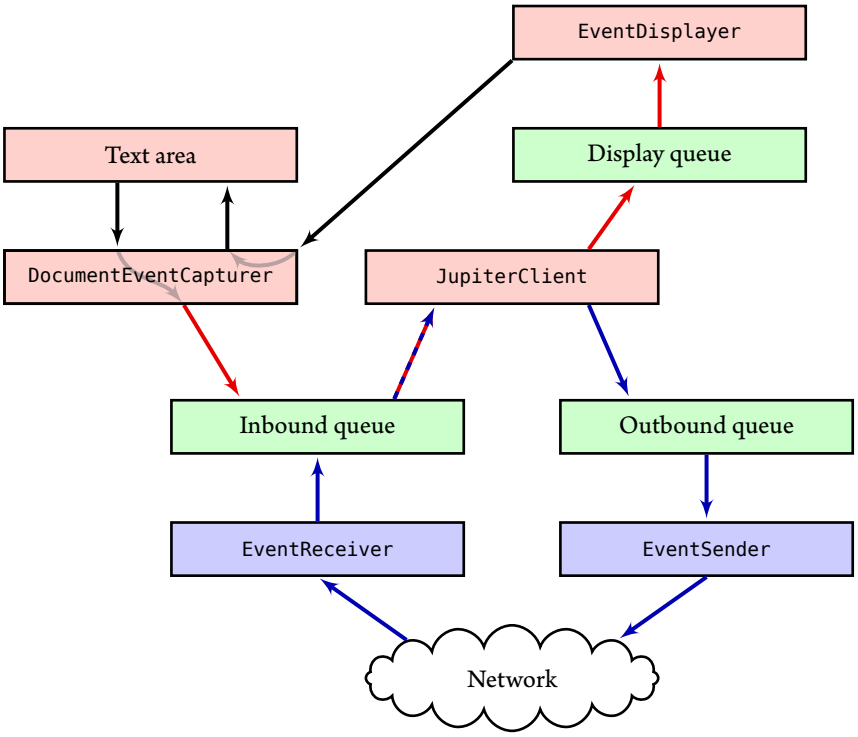
This is most of what the Jupiter algorithm does. It gets only slightly more difficult when client and server diverge by more than one operation.

# 3 Code Overview

FiXme: The EventDisplayer communicates directly with the text area.

Thanks to the layered architecture (not very stringent) and message queueing, we didn't have to change much of the existing code in order to allow for distributed editing of the same text. – Of course, one text area had to go and `DocumentEventCapturer` and `EventDisplayer` (formerly `EventReplayer`) now operate on the remaining one. But otherwise only the `JupiterClient` got plugged in between receiving and sending, and capturing and displaying events, respectively. Figure 2 shows the new configuration.

Figure 2: `Event`'s path through the system in the editor as provided and in our version. → denote `TextChangeEvents` and → denote `JupiterEvents`.




## 3.1 `JupiterClient`

The `JupiterClient` is called like that because it will be used as the client Jupiter class for the next handin with $n$ parties. Right now, there is one `JupiterClients` on each of the two editors running.

All events go through the `JupiterClient`: Both the user inputs in the form of `TextChangeEvents` and events from the other `JupiterClient` in the form of `JupiterEvents` come in through the inbound queue. If an incoming event came from the user's actions on the text area, the `JupiterClient` wraps it in a `JupiterEvent`, including a timestamp and sends it to the other editor. It gives it to the `EventReplayer`, so that the user actually sees the result of her input. If, on the other hand, an incoming event came from the other editor, the `JupiterClient` transforms it, if necessary, before giving it to the `EventReplayer`.

## 3.2 `DocumentEventCapturer`

It is not possible to send the text directly to the text area from the `EventDisplayer` without it going through `DocumentEventCapturer`, witch extends `DocumentFilter`. To circumvent this we put a switch in `DocumentEventCapturer`.

Before `EventDisplayer` send the text to the text area, it set the `isGenerateEvents` to **`false`** to change the function of `DocumentEventCapturer`. When it is set it does not generate event, but instead add text to the area. Afterwards it it set to **`true`** again.

The gray arrows on figure 2 represent this switch. The listing below shows how the switch is implementet in the `insertString`-method in `DocumentEventCapturer`.

File: DocumentEventCapturer.java

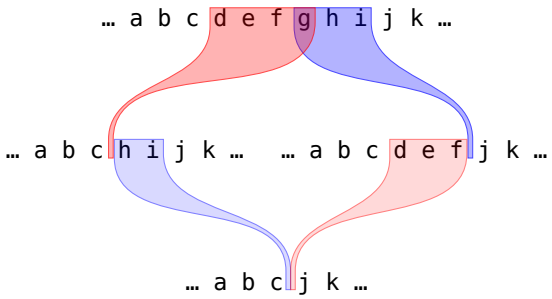
```
41      public void insertString(FilterBypass fb, int offset,
42                               String str, AttributeSet a)
43      throws BadLocationException {
44      /* Queue a copy of the event or modify the textarea */
45          if (isGenerateEvents) {
46              eventHistory.add(new TextInsertEvent(offset, str));
47          }
48          else {
49              super.insertString(fb, offset, str, a);
50          }
51      }
```

## 3.3 `Transformer`

As mentioned earlier, the transformation function can be implemented in different ways. The Saros project, for example, uses what is described in [6]. We decided for a more primitive approach, which covers all cases, but does not always yield satisfactory results. However, we wrapped the function `transform` in a class `Transform`, so that the structure can easily be changed to match the strategy pattern. Thereby, different solutions to the same problem could be plugged in without effort.

FiXme: Possibly needs to be adapted to the text. Or just include an ASCII art thing like I drew in the Transformer.

Figure 3: Transformation of two overlapping remove operations




As an example, figure 3 shows what `transform` does for two overlapping remove events: On the local machine the smaller removal to the right and on the remote machine the larger removal

to the left have happened. In this case, transforming the operations amounts to make them delete less and possibly in a different place. The following listing shows the piece of source code that covers this and some other cases. The last two assignments are the transformed operations in this case.

File: Transformer.java

```
149         else if (recTE instanceof TextRemoveEvent
150                 && locTE instanceof TextRemoveEvent) {
151             TextRemoveEvent recRemove = (TextRemoveEvent) recTE;
152             int recOffs               = recRemove.getOffset();
153             int recLen                = recRemove.getLength();
154 
155             TextRemoveEvent locRemove = (TextRemoveEvent) locTE;
156             int locOffs               = locRemove.getOffset();
157             int locLen                = locRemove.getLength();
158 
159             if (locOffs > recOffs) {
160                 if (locOffs >= recOffs + recLen) {
161                     transRecTE = new TextRemoveEvent(recOffs, recLen);
162                     transLocTE = new TextRemoveEvent(locOffs -recLen, locLen);
163                 }
164                 else { // Removals overlap
165                     // rec contains loc
166                     if (locOffs + locLen < recOffs + recLen) {
167                         transRecTE = new TextRemoveEvent(
168                                         recOffs,
169                                         recLen - locLen
170                                     );
171                         transLocTE = new TextRemoveEvent(0, 0);
172                     }
173                     else {
174                         transRecTE = new TextRemoveEvent(
175                                         recOffs,
176                                         locOffs - recOffs
177                                     );
178                         transLocTE = new TextRemoveEvent(
179                                         recOffs,
180                                         locOffs + locLen - (recOffs + recLen)
181                                     );
182                     }
183                 }
184             }
```

# 4 Conclusion

We had an editor whose instances could connect to each other and where the user could see what the other was doing to her text. Now we extended this, enabling both users to edit the same text at the same time. For synchronisation we use the well-known Jupiter algorithm, one advantage of which is that users see their changes without delay.

As usual, some issues, mainly in regard of user friendliness remain. For instance, there is no indication whether editors are connected. However, those things are in no way related to distributed systems and therefore have to stand back behind things that are. Another example is the relative primitivity of the `transform` function. This is also not severe, since it only very rarely, if at all, affects sensible text editing actions.

All in all, the architecture created for the previous version of the editor proved quite successful in letting us concentrate on the synchronisation issues central to this exercise. And using the exising Jupiter algorithm is, in our opinion, a much better approach than inventing some half-baked own synchronisation method.

# A Finding the Code and Running the Editor

The file `Code1864-ex11.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`.

FiXme: BibLaTeX might be displaying too much information. Just throw them out of literatur.bib.

# References

[1] *Gobby. a collaborative text editor*. 0x539 dev group. Jan. 15, 2014. URL: `http://gobby.0x539.de/trac/` (visited on 02/22/2014).

[2] *Google Wave Protocol*. URL: `http://www.waveprotocol.org/` (visited on 02/22/2014).

[3] David A. Nichols et al. "High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System". In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. UIST '95. Pittsburgh, Pennsylvania, USA: ACM, 1995, pp. 111–120. ISBN: 0-89791-709-X. DOI: 10.1145/215585.215706.

[4] *Real-Time Distributed Software DevelopmentSaros*. Saros. 2014. URL: `http://www.saros-project.org/` (visited on 02/22/2014).

[5] Daniel Spiewak. *Understanding and Applying Operational Transformation*. May 17, 2010. URL: `http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation` (visited on 02/22/2014).

[6] Chengzheng Sun et al. *Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems*. Mar. 1998.