Distributed Systems

Exercise 14

Richard Möhn (201311231) Mathias Dannesbo (201206106)

March 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 another version of our distributed editor. This time it is not only able to synchronise editing actions between two running instances, but arbitrarily many (within the bounds of the computers' resources and some number representations). For this, we didn't need to change the Jupiter algorithm, but rather had to add lots of infrastructure.

The report describes how we enabled the editor to do n-way synchronisation and our efforts to implement the management of the distributed system. It discusses some of the decisions we made in developing the new version of the editor. The conclusion contains a list of issues with the editor that still need to be addressed.

2 Code overview

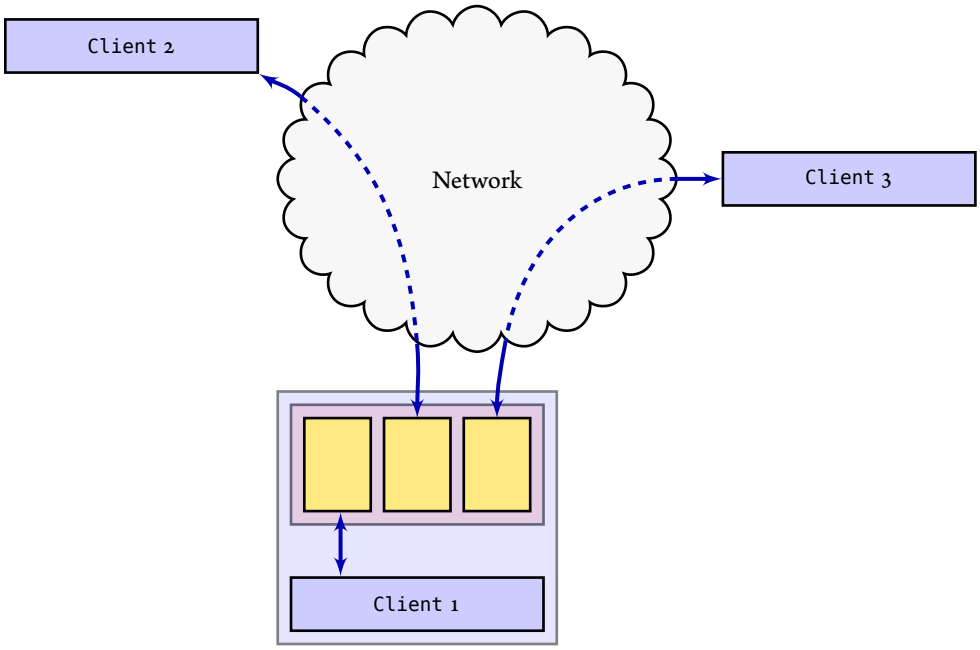
For n-way synchronisation to work the Jupiter algorithm requires a central coordinater (called server) to which all editors (called clients) connect. One of the instances of the distributed text editor runs both the server and the client in different threads.

The changes to the previous version can be summarized as follows. We separated the Jupiter algorithm from the event processing by splitting `JupiterClient` into the classes `Jupiter` and `ClientEventDistributor` and `ServerEventDistributor` for the client and server respevtivly. We introduced the classes `Server` and `Client` taking care of the server's and clients' `EventReceivers`, `EventSenders` and `EventDistributors`, thereby pulling out code from the main class `DistributedTextEditor`. We created `ClientHandle`, which the `ServerEventDistributor` uses as an interface to the clients it manages. Also

the process of moving the server between clients (described later) required several new `Events`.

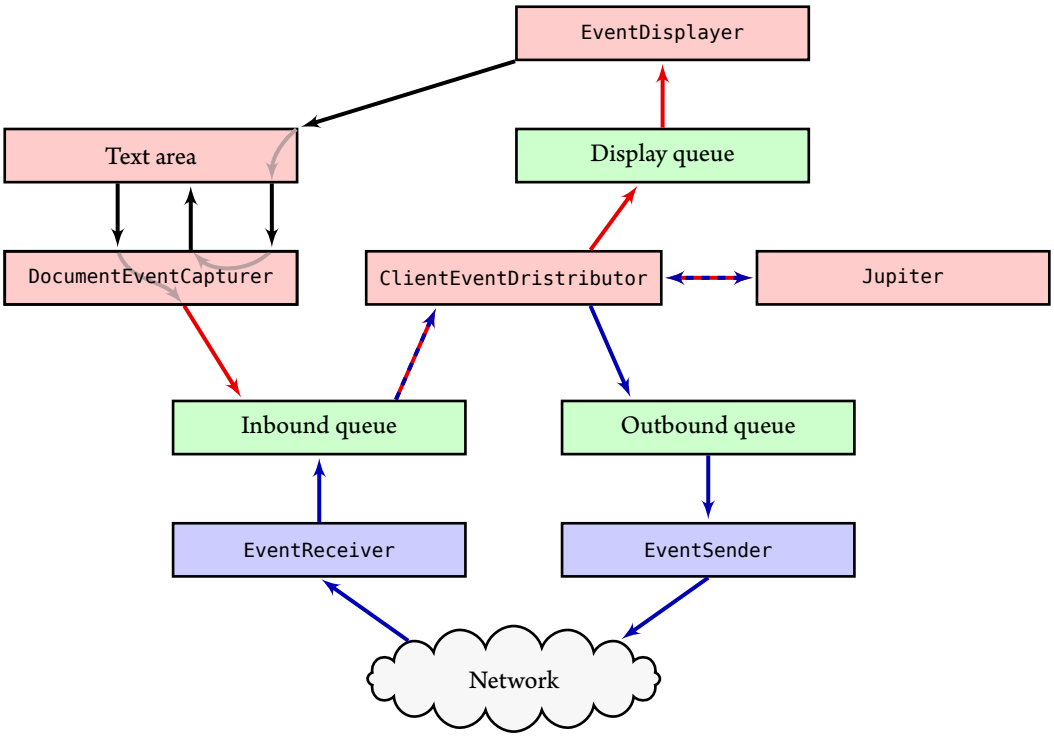
3 *n*-way synchronisation

Figure 1: The whole system. `Client 1` runs a server: ▭ denote the `Server`-object. It has a `ClientHandle` inside for each client. This is denoted by ▭. Details about the server are in figure 3. Details about the client are in figure 2.



The Jupiter algorithm doesn't need to change in order to accomodate *n*-way synchronisation [1]. Instead it uses *n* instances of two-way synchronisation, handled by two instances of the Jupiter algorithm each. The Server has one `Jupiter` object for each client. When the `ServerEventDistributor` receives a `JupiterEvent` from a client, it uses the `Jupiter` instance corresponding to that client to transform it. Then it propagates the resulting operations to the other `Jupiter` instances which then synchronise with the `Jupiter` instances on the clients they are responsible for. Only after one event is thus received, transformed and broadcast, the next event can be processed. This sequentiality makes special synchronisation measures between the `Jupiter` instances on the server unnecessary. Note that the number of simultaneous edits is only limited by computational and network resources available at the server.

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



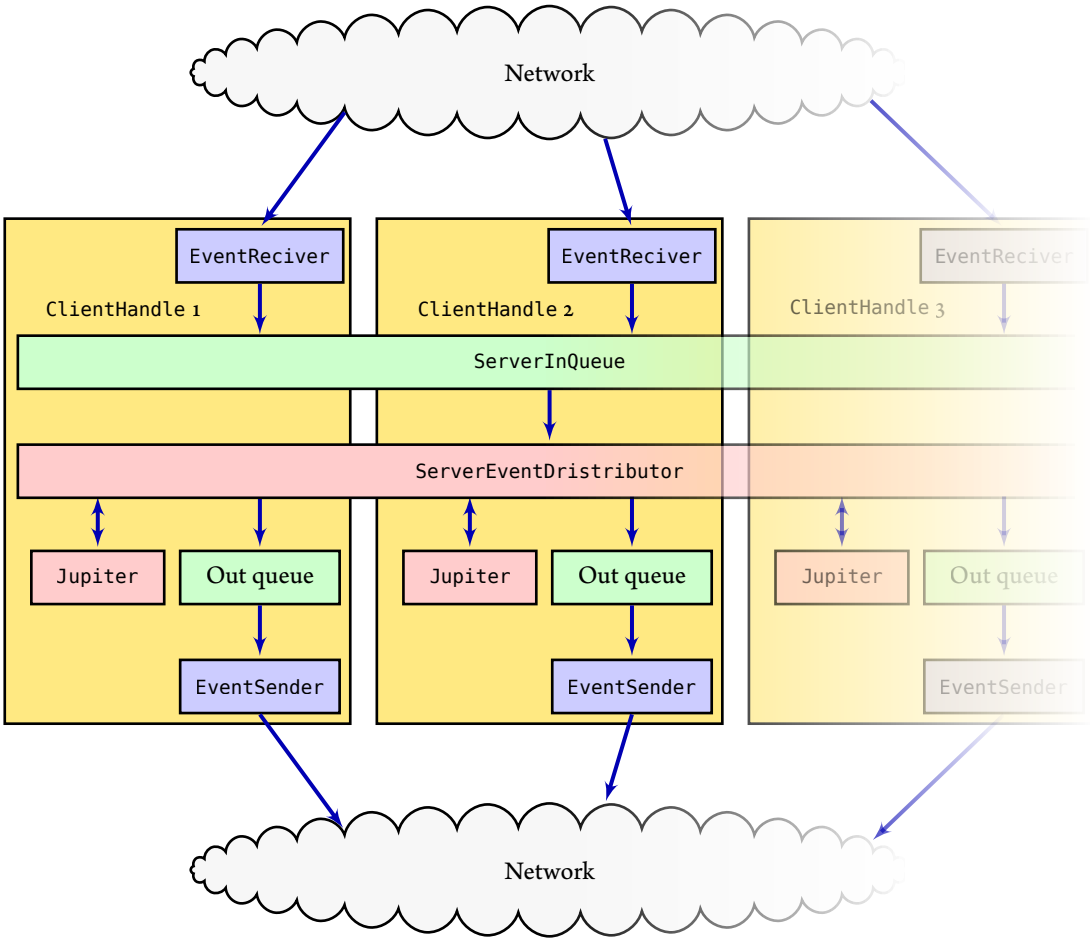
Figures 1, 2, 3 show an overview of the processes, the client (which didn't change much) and the server with the `ServerEventDistributor` handling some clients.

4 Peers joining

The server continuously listens for new clients on the socket. When a client connects, the `ServerEventDistributor` creates a `ClientHandle` for it, sends the current text to the new client and after that everything goes on as before. – The new client is immediately able to make changes of others and see their changes and the other clients won't notice that someone new has joined the editing.

It was also required that an editor should be able to join the network by contacting any of the editors part of it. We didn't have time for that (see section 6), put we would have done it like this: also clients continuously listen on some port. When an editor B contacts a client A in order to become part of the network, A responds with information about the current server. Then B contacts the server according to these information.

Figure 3: Objects in a `Server`-object. Each `ClientHandle`, which is denoted by ▭, is responsible for a `Client`. They each have an `EventReciver`, `EventSender`, out queue and `Jupiter`. They all share a single `ServerInQueue` and single `ServerEventDristributor`.



Note that the peer running in the same process as the server communicates with the server also through sockets rather than through a local data structure. We decided to do it this way, because it enables us to treat all clients the same.

5 Peers leaving

As long as peers that are only clients leave, everything is fine: the connection is shut down in a similar way as before and the server removes the `ClientHandle` from its records. However, when the peer that also runs the server disconnects, things get complicated, because the server has to be moved to another peer and the other clients have to be made aware of that. `ClientEventDistributor` and `ServerEventDistributor` therefore not only process `TextChange`- and `JupiterEvents`, but also contain a state machine each for managing the process of moving the server.

It is not feasible to describe these state machines with prose and there was no time to create proper digital pictures. However, this is roughly the procedure: The client the editor that also runs the server (called old server) disconnects. The old server and the other clients process the pending events and stop at a clean common state. The old server causes one client to start up a new server, the other clients disconnect from the old server and connect to new server.

The users don't notice this process, because their inputs are buffered and sent to the new server when everything is stable again. This contributes to our generally high degree of distribution transparency (at least as long users operate within the specified bounds).

6 Open issues

We slightly underestimated the amout of work and as a result, hadn't time for resolving some issues and doing proper cleanup. The following is a list of things we failed to do.

- Required feature: It was demanded that an editor can join an editing session by contacting any of the existing peers. In our implementation it can only do so by contacting the server. See also section 4.
- Proper handling of exceptions: The whole connecting and disconnecting process is much more complex than before which makes it quite easy for the user to press the wrong buttons at the wrong time. On such occasions the editor always crashes, because we want things to fail dramatically instead of ceasing to work without obvious reason.

  Worse with respect to the subject of the course – distributed systems – is that we also crash on problems caused by the communication between the clients.

This is not quite appropriate, because we're distributed systems are generally not reliable, and means that the degree of failure transparency is rather low.

- There is an exception that existed since the first version of the editor and the cause of which we haven't yet found: When clients connect again after they disconnected, an `IllegalArgumentException: Invalid remove` is thrown. However, since the `EventDisplayer` just catches and prints it and it doesn't appear to have any negative effect, we just let it be.
- Sometimes with wild concurrent editing things go wrong and wrong operations are attempted to be applied. We have no time to investigate this.
- Moving the server between different physical machines doesn't work properly. – Either we have misunderstandings of network programming or our protocol for moving the server is insufficient, resulting in timing problems that don't occur when trying things locally. Note, though, that we put quite some thought in the protocol and strived to make it correct (in a distributed and nonsequential systems way). But it is complicated enough that we might have overlooked something.

7 Lessons learned

Our systems has arrived at a state where, were we to develop it further, throwing it away and starting over might be considered. This is not because it has become entirely unmodifiable, but rather because we have learned so many things that the necessary and sensible refactorings would be quite extensive.

One of the main issues complicating development is that there is no separation between the code managing the connections and the code responsible for synchronisation. These clearly belong into two different layers. That was no problem in the previous versions, because barely any connection management was required. Now, however, this part of the program is substantial and should be well hidden.

Related with this is our choice of the synchronisation algorithm. – The Jupiter algorithm is derived from an algorithm which does n-way synchronisation without a central coordinator [1]. The article claims that that made it much simpler, but this might be a tradeoff between simplicity in managing clients and simplicity in synchronisation. It clearly makes sense for them and the other applications mentioned in the last report to go this way, since they use a completely server-based approach. However, we are required to move the server around, which makes things different. We should have investigated this more thoroughly.

A major source of program complexity is closing threads and sockets properly when clients disconnect. Since this shouldn't be difficult, we were wondering whether there

are better ways to do this in Java. Maybe the way we use threads and make them communicate through queues is not idiomatic. Maybe it's attempting to program Go in Java? If that is the case, it wasn't by intent, but lack of knowledge of proper Java programming.

8 Conclusion

We had an editor whose instances could connect to each other and where two users could edit the same text at the same time. Now an in principle unlimited number of users can edit the same text at the same time in a transparent way. For synchronisation we still use the well-known Jupiter algorithm without fundamental changes. – It can easily be utilised for n-way synchronisation.

By far the greatest difficulty and largest amount of work was in managing connections and disconnections between clients and server and in moving the server from one client to the other. We are sad to say that it kept us from doing quite some necessary things.

All in all, however, our concept worked out quite well, the remaining issues are manageable and we learned a lot.

A Finding the Code and Running the Editor

The file `Code1864-ex14.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`.

References

[1] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. "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.