Report 3
Distributed System

Hannibal Krabbe-Keblovszki, 20102295
Morten Johansen, 19870213
Fie Hebsgaard, 20103511

March 2014

1 Introduction

For this hand-in we had to extend the distributed editor from last hand-in. In the following sections we are going to explain how we created a distributed editor with more than two peers. Firstly we are going to explain how we edited the whole system from working with vector clocks to working with Lamport numbers and furthermore how the system now is based on the scheme chord idea. After this we are going through the systems functionality and how this is done and at the end we ends it all with a conclusion. And here we also have to say that we have all three an equal percentage of the work there has been done in this hand-in.

2 The network structure

In this hand-in we have decided to model the system on a scheme chord inspired network. With this we created every node to have both a server and a client side, such that the last hand-in was not in vain. In the following section it is spelled out how the scheme chord is used and implemented. This was done because it seemed as a simple solution with only one client/server(most of the time) pr peer.

2.1 The scheme chord inspired algorithms

Every node is created with a client side, a server side and a nodenumber (the nodenumber to ensure the priority of textevents – more on this later)

When the network is created, we start the system by making one of the editors start its serverside. After this another node starts its serverside, and connects its clientside to the first server. Furthermore hereafter this first node creates its clientside and makes a connection to the other node. The nodes also keep a counter to know have many nodes there is in the network. This is done to make sure every node get the right number of acknowledgments of each events (explained in the next section).

When a new node wants to be added to the network, the client side of this node, ask for permission to one of the nodes (serversides) in the network. When this permission is sent, the node in the network sends a request to its successor to make this node change its successor to the incoming node. It creates a new client that makes the connection, and close down the old client.

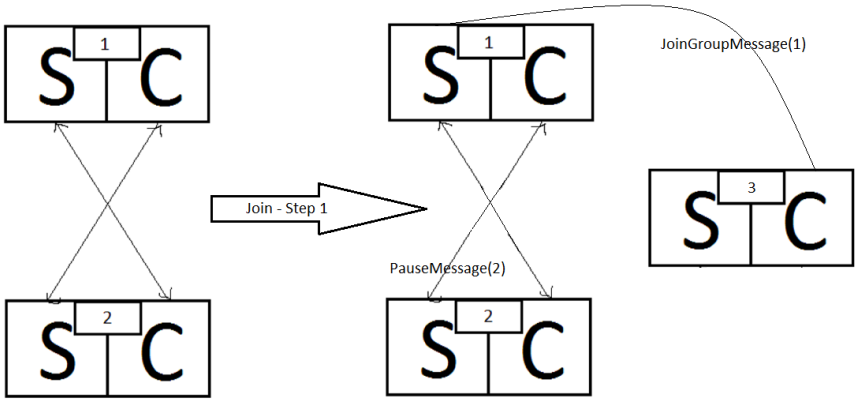
To make the connection possible and make sure that the incoming node can see what is already written on the others screens, we lock all the text editors for a few seconds, to make the connection between the new node and the network and for emptying the queues so they are ready to handle the new peer.

We take advantage of that we have locked screen to change the text on all editors, to be completely sure, that everybody has the same text as a starting point, when a new peer is joining. We also clear all the tables with text input, in the time the system is locked to be sure that editors starts at the same point.

When a node leaves the network we do almost the same as when we make a connection. We lock the screens, so the two nodes connected with the one leaving, has time to update their new successor and predecessor. If we doesn't lock the editors, we get a problem that is that someone types before the connections are made which result in desynchronized editors.

2.2 Implementation of the "scheme chord"network

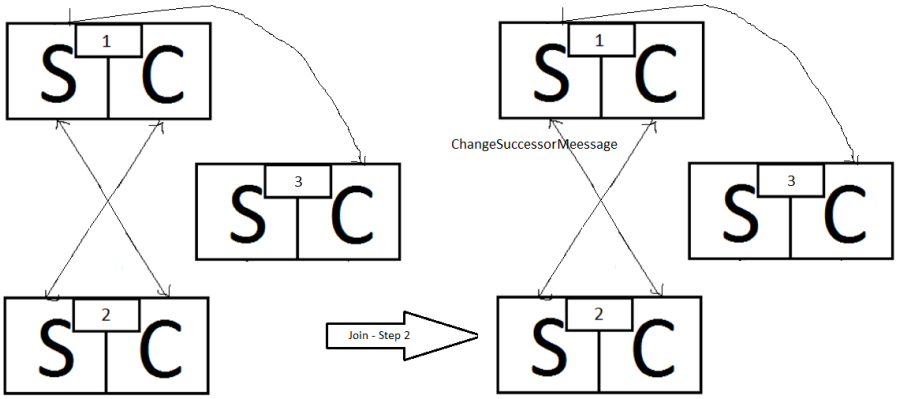
When a new node (here called 3) is joining the network it needs to have started its serverside, before requesting for joining the network. When a node (here called 1) in the network receives a join-request, which is the first thing that happens, node(1) sends out a broadcast-message (which here is a pause-message) to all the other nodes to lock their screens. This step is shown in the picture below:



Node(1) now sends a change-successor-message to node(2), which is node(1)'s successor, that starts a thread that sleeps for a few seconds. It then increases the number of nodes. When node(2) receives the changeSuccessor-message it changes its successor and the number of nodes directly by calling:

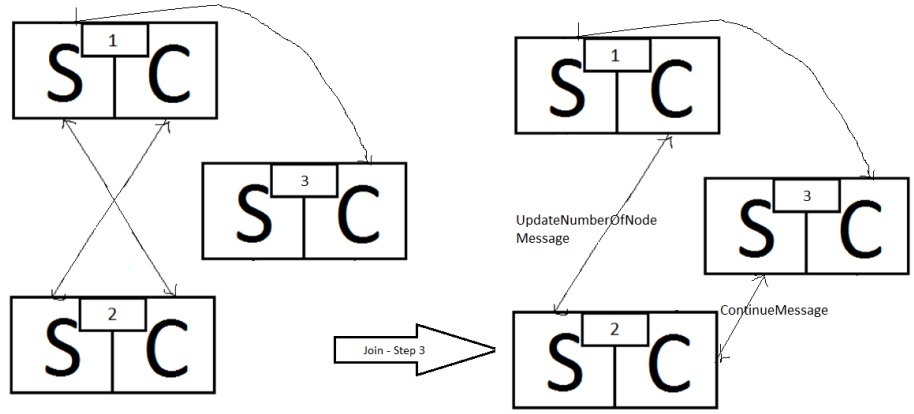
```
suc = changeSuccessorMessage.getNewSuccessor();
numberOfNodes = changeSuccessorMessage.getNumberOfNodes();
```

To see where the node actually sends the message just look at the next picture:



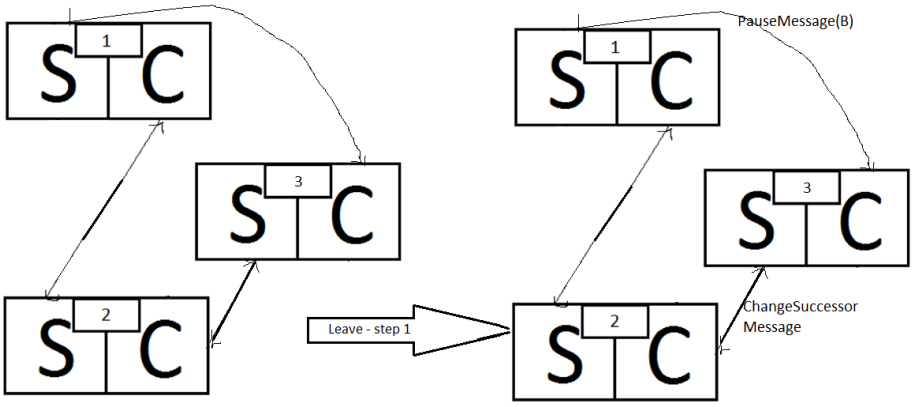
Then node(2) now creates a new client and connects to node(3) (while the old to node(2) is still running), when the connection to node(1) is requested, the old client is closed and the networks gets a few second to make the new connections, before a continue-message is broadcasted to everybody in the network and the editors can used again. this continue-message contains the text field from node(1) and everybody is updated with this text, and a message in the

bottom area says, that it is possible to write something again. All this happens as in this picture:

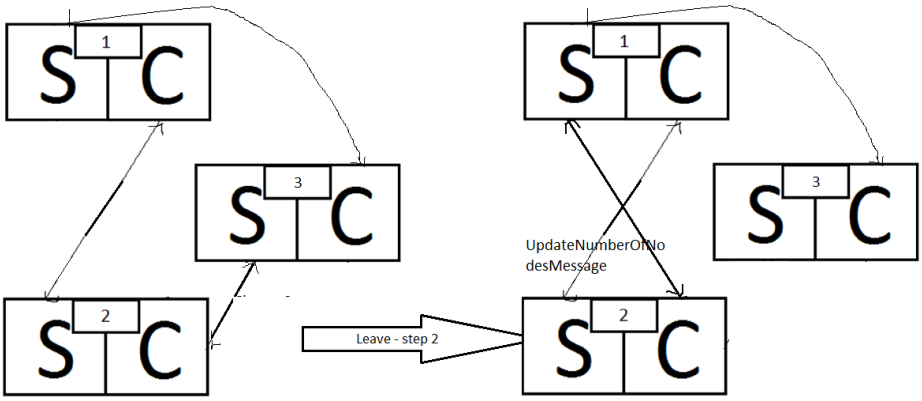


So actually we have made the join now.

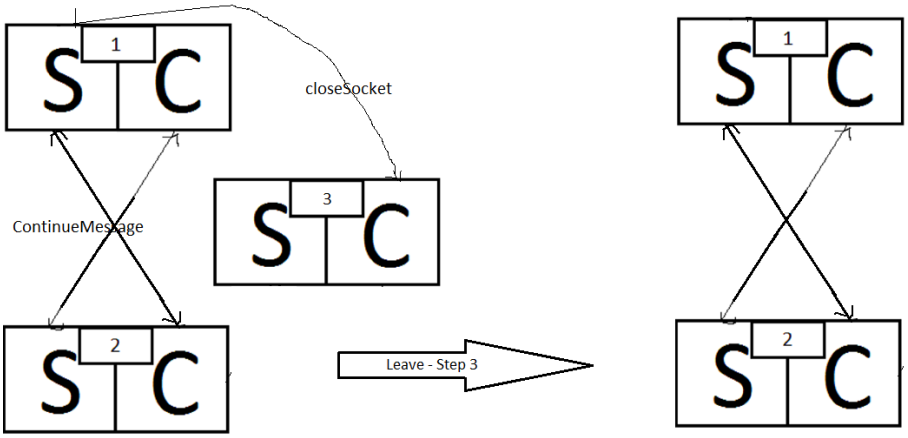
It is now time for how the disconnecting thing works in this system. When a node wants to leave the system it has to disconnect. In the picture below the node(3) are leaving. When this happens it sends a change-successor-message to node(2) and after this it broadcasts a pause-message through the client (in the picture the broadcast is shown with a B):



After these changes the node(2) sends an update-message through the server which updates the whole system and this is done while it is all paused so it is save to do without thinking of the synchronization. One can see the meaning in the picture below:



For the last step in the leaving method we just want to get rid of the node(3) that wants to leave. So here we just close the client socket for node(3) and node(2) then sends a continue-message to the rest of the system so we can continue typing in the editor.



This is the leaving part of the system. It can be hard to keeping track of the message parsing so we just make a little section explaining what each of the messages does in the system.

2.2.1 Message parsing

We have implemented a general networkMessage class, that have a sender and receiver address, so we could make a lot of different messages and distinguish between them. Some of the messages is sent around the whole system and some of them is just sent for a single peer. Below it is explained how all these message works individually.

The messages in alphabetical order:

ChangeSuccessorMessage When a message of this type is sent, the node sending it, is sending the message from a server to its client, containing the new successor name and the new number of nodes the network should be updated to, besides the nodes own servername (sender) and the clientname (receiver).

When a node receives a ChangeSuccessorMessage a lot of changes is made, and the system is already locked. First of all, the nodes successor is set to be the new address from the message. A new client is created after this, making a connection to the new servername. It starts the client thread and close the old client connection.

Now we start a timer to keep the system locked, until all the right connections are made. We do that by starting a thread that sleeps for a few seconds, and then sends a continue message.

ContinueMessage This message should also reach all the peers in the network. We are sending the same thing as in the join-message but we are also sending the text from the editor so it is possible to update clear an update all the editors with the same text so they can start from the same starting point. Furthermore we resets the table with the textevents so that the editors are starting with the same textevent-table.

JoinGroupMessage When a peer receives a message of this type it sends a pause-message to the next peer in the scheme chord and this message stops when it reaches its sender. This means that all the editor on each peer is now frozen for a while. This is done so it is possible to change the successor without any conflicts. The scheme chord should in this way be updated and as explained in the ChangeSuccessorMessage section we ensure that the system is awake again after the change of the successor (with the continue-message).

PauseMessage The pause-message is yet another message that should be received by all the peers in the system. So what this message does is just that it disables the feature to write in the top text field at the text editor. With this we ensure that there isn't any input to the text field while a new peer is connecting to the system. We need this to ensure the synchronization of the text editors.

UpdateNumberOfNodesMessage This message is sent to keep track on the number of nodes connected to the system so that the systems knows how many acknowledges it should have received before writing something in the text field. If we haven't created this method we couldn't ensure the synchronization of this system. Furthermore does this message update which server a client should connect to if there has been a connect or a disconnect in the system.

3 Lamportclocks and multicast

In this sections we will explain and discuss how the Lamportclocks and the multicasting is used in general. Especially are we going to explain how the whole thing is being ordered.

3.1 The algorithm overview

We chose to use Lamport clocks and sending acknowledgments, instead of vector clocks. The argument for this was to get all editors to show the same text, which we did not succeed in doing for the second hand-in where we used vector-clocks.

The Lamportclock is used to make totally ordered multicast, which should get all editors to show the same text.

All peers have a number which they increment with 1, every time they create a textevent(insert or remove text). This number is send to all other editors in the network, when a textevent is send. When an editor receives another editors textevent, it compares its own counter(Lamportclock) with the received counter and sets its own counter to max(own counter, received counter)+1. This is then used to order all events according to the Lamportcounter of the event in a list, and if 2 events have same Lamportclock, they are ordered according to the editor name. When inserted into the ordered list, the editor checks whether any events originating from this editor is in front of the newly inserted event. If not then the editor sends an acknowledment to all other editors. If an editor have received acknoledgement from all editors for the event at the front of the list, it is removed from the list and printed on the screen. This should secure that all textevents are presented in the same order at all editors. Now we have keep a list of all events in the right order to calculate change in the offset for each event. To prevent that we need to keep all events back from the beginning of time, every editor keep a note(Lamportcounter and editorname) of the last event printed before the actual event that is about to be printed. This ensures that we only need to keep events from the editor that send the last event, that has a Lamportclock higher than the last printed event.

3.2 Implementation

We have made a LamportCounterObject, so when every distributed texteditor is started, then the node, the DocumentEventCapturer(here called dec), and the serverside also is created with the same LamportCounterObject. Whenever a textevent happens in the screen, the "dec"makes sure that the counter is increased by one, and then sending a textevent message out to every node in the network. The message contains both this textevents LamportCounter, the

node name, the offset and the last seen Lamportcounter and last node name in the latest textevent displayed. There is a message for both insertEvents and removeEvents, that contains either the string or the length of what should be inserted or should be removed. All textevents either made by dec or received from other editors are placed in the queue "eventHistory"located in dec. This queue is emptied by eventReplayer.

The serverside is the one to see all the messages first, and if it sees a textevent then it sets its LamportCounter to max+1. Max is the highest number of lamportCounts yet seen. The server then sends the textevent to "dec", and further through the network if it isn't its own name.

The eventReplayer gets all the textevents from the dec and it is this job to handle the text is inserted correct on the screens. When a textevent is recived by the eventReplayer, it is put in ordered list "priorityList"where the events are ordered by LamportCount and after that "nodeName", this means that the oldest events are first in line. The eventReplayer also makes some checks to see if the textevent should be acknowledged or wait until another events has all its acknowledgments. If the textevent has all its acknowledgments then the eventReplayer start to calculate the offset of which the event should be shown on the screen.

All the textevents that are written on the screen is put in a queue so we can calculate the offset. We clean out in this queue, so it doesn't get too long. These are placed in the list ordered by the LamportCounts.

When the eventreplayer receives an acknowledgment from the network, the eventreplayer will run the updateACK methods that finds the textevent that is being acknowledged, and check if the textevents has received all the necessary acknowledgements (which is the number of nodes in the network-1). If so, the textevent will be shown on the screen and deleted from the "priorityList". If the textevent is from the node reading it, it sends out an acknowledgment for this package to the network.

When the offset of a specific textevent is being calculated, the first we check is the LamportCounters or the name in the queue is different from the LamportCounter or the name in the textevent respectively. If one of them is different, and the offset of the text event we wants to caculate is greater or equal to the one in the list, we calculate a new offset. This make sure it only changes the offset if someone is typing.

4 Extra features

For this hand-in the extra features are that when you type your nodenumber, portnumber and ipaddress one can only listen in the menu and after this you

can only connect. This is done because you have to do the setup for the system in this order (when each editor is both a server and a client).

Furthermore we have added the extra feature that when one click at the nodenumber text, portnumber text or the ipaddress text is will erase this text so one can just type in the data for the text field without erasing the existing text. This feature has been very delightful because we have tested this system a lot – This means that we have saved a lot of clicks on the delete button.

5 How to run the system

1. Choose a llama-machine.
2. place all java-files in a directory of your choice.
3. While in this directory write "javac *.java"to compile.
4. Still being in the directory write "java DistributedTextEditor"to run the program.
5. Go to the inputfield with the text "Port number here"and type a port-number between 1026 and 65355. This will be the port where the peer will listen, and has to be used on all peers that will be part of the network.
6. Go to the inputfield with the text "NodeNavn"and write an integer, that will be the name of this editor. The names of the editors must be different from each other.
7. Open the menu "file"and choose "Listen". You will now have created the first peer in the network.
8. Choose another llama-machine and go to the directory where the program files where placed.
9. Write "java DistributedTextEditor"to run the program.
10. As for the first machine fill in inputfield "NodeNavn"and "Port number here". Remember that port number has to be identical for all peers.
11. Go to inputfield with the name "IP address here", and write the ip-address of the first machine, which is shown on the first machine in the title of the program, like this "I'm listening on 10.10.10.12:57354". **When chosing a peer to connect to, be aware that it is only possible to connect twice to a peer for some mysterious reason. This means that only one other peer can connect to a peer in the network**

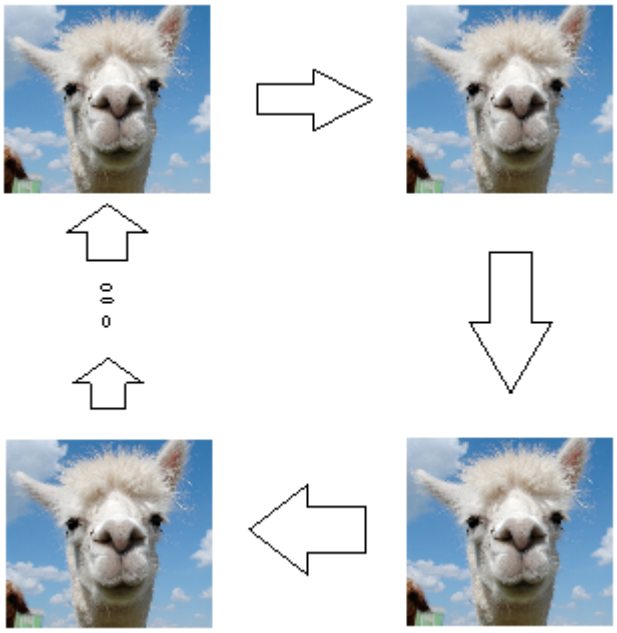
12. Open the menu "file"and choose "Listen". You will now have started the server part of this peer.

13. Open the menu "File"and choose "Connect"to connect to the first machine.

14. It should now be possible to write on one peer, and it should be shown on the other peer.

15. To add additional peers perform step 8-13. **When connecting the editors will be locked for some seconds, so it is not possible to write at any editor in the network, while a peer is connecting or leaving.**

16. For an editor to leave , open the menu "file"on the editor that should leave and choose "Disconnect".



6 Conclusion

This hand-in has been challenging and time demanding. The solution meet the requirements from the description with some reservations. The system always shows the same text on every screen (that is what we have experienced) but when you're deleting very fast and typing in an different area at the same time the result might not be intuitive. That we have to lock the entire system when

connecting/leaving might not be the most optimal solution, and this is one of the places where we would make changes if we had the time.

Furthermore it is now possible to connect to the first node and after this you can connect again, but one can't reconnect to the same node again. If one wants to join the system they have to join on a different peer (that has not been in use yet to insert another node in the system/network) than the initial one. When you connect to another the one after this can't connect to that one and so on. If we had had the time we could have made this working.

With this it is still important to say that is it possible to create a big network with this system. One should just do it the right way.