

# DM883 Distributed Systems: Project II

Andreas Lyngholm Poulsen  
andpo17@student.sdu.dk

Vicki Mixen  
vimix17@student.sdu.dk

Anne Møller Madsen  
anmad17@student.sdu.dk

Assigner:  
**Marco Peressotti**

---

May 27, 2021



# 1 Introduction

We chose, for our final project, to create a decentralised chat. The chat was to support multiple groups and allow users to join these to communicate with each other. The system had more requirements as well. The system was to be able to list users in a group as well as searching for users based on their name. The ability to search for groups based on their name. Causal ordering of messages and a message history, which is bounded by size or date. A user interface for users to use the chat was also implemented and a requirement.

## 2 Architecture

We took a starting point in the algorithm, Chord, as we wanted to design a decentralised chat in Erlang. Chord is an algorithm that creates a ring and uses successors to communicate within the ring. During the lectures, a version of Chord was implemented. This is what we took as a starting point. We chose to use the implementation of Chord with the simple routing table, because it was the easiest to continue adding features to. This version though, was not fault-tolerant; this we had to implement ourselves. For our architecture to be fault-tolerant, we introduced a list of successors. This resulted in a lot of challenges which we are discussing in the subsection 2.3.

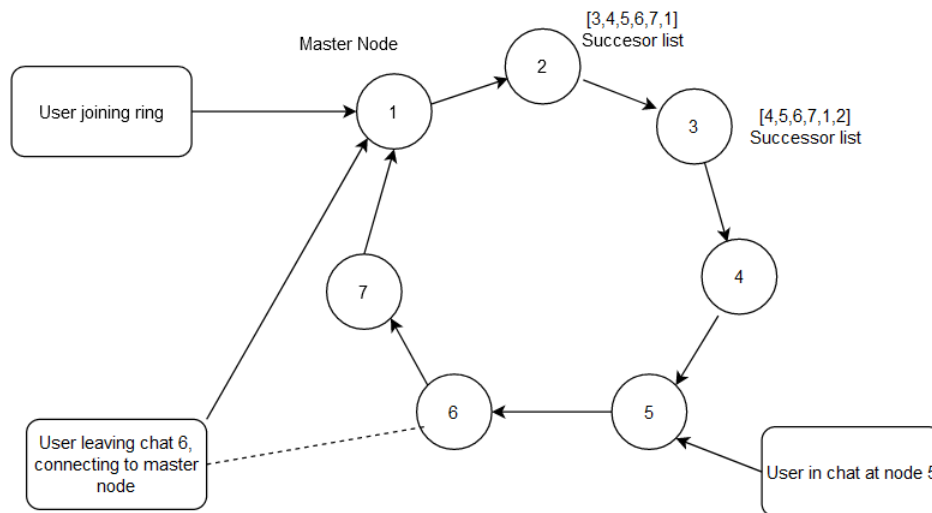


Figure 1: The architecture of the chat.

We wanted a fully decentralized system, but found this to be very difficult. We decided to go with what we believe to be semi-decentralized. We have a "start node" which we are introducing when the system is booted. This node is contacted when a new node is introduced to the ring. We believe that this makes it semi-decentralized.

For the ring, we have a single ring with multiple nodes, where each node is a chat in our system. With our design of the chat, where we use only a single ring and interpret each node in the ring as a chat with a list of users and messages, the features required in the project becomes straight forward to implement. When a user wants to create a new chat, we simply add a new node to the ring. This is how our system is thought to work. We can easily delete a chat, simply by removing it from the ring. Every time we are making changes to the ring, we are stabilizing it to make sure everything is correctly setup and stable. A diagram of the architecture can be seen in Figure 1

Due to the nature of the stabilize function, we need to stabilize the network all of the time to ensure that all of the nodes are correctly connected. The data is only accessible when the network is stabilized. We had some issues with some data not being accessible due to the network not being stabilized. To combat this, we simply decreased the stabilize interval from 1000 ms to 100 ms. The stabilize function runs a lot more, but we can ensure that the network is stable most of the time.

As mentioned earlier we had to introduce a list of successors for each node in the ring for Chord to be failure tolerant. This list is created and used in stabilize, such that if stabilize notices that its successor is not responding, it can find its next successor in the list. This can be seen in Figure 2. This way, even if a node crashes, the rest of the ring continues to run. The ring is however only tolerant of up to 8 nodes failing in the system, as this is the number of nodes in the successor list. We chose this number, as it is the size of the keys for the nodes, and this was recommended by the lecturer and the Red book. If the system became large enough to have to use larger keys, the system would of course be tolerant of more node successively failing, because the number of successors in the successor list would be incremented to fit the key length.

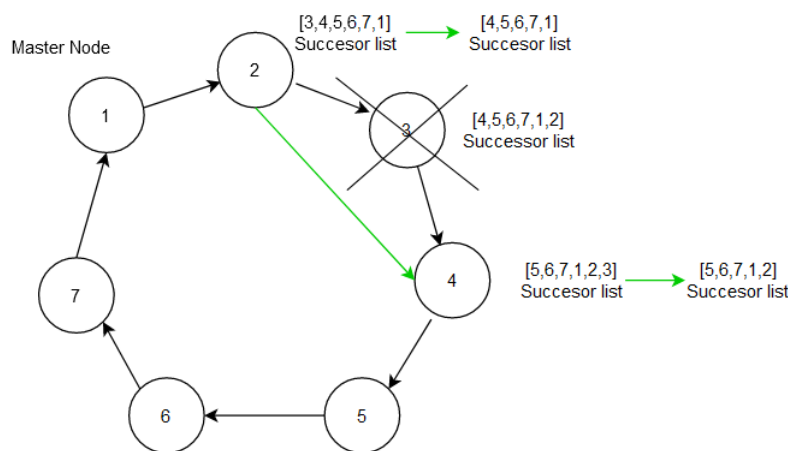


Figure 2: Simulation of node 3 dying

When a user wants to join a chat, the user is simply put in contact with the node in charge of the chat. The user is then added to the list of users of the node.

```

1 % This opaque is simply used to hide the fact that our users are simply a list.
2 % We make sure to only append user records the list to use it as a list of User.
3 -opaque(users() :: list()).
4
5 % A node is a group.
6 -record(node,{
7     name = [] :: string(),
8     key :: pid(),
9     pid :: key(),
10    users = [] :: users(),
11    messages = [] :: q_entries(),
12    temp_q = [] :: q_entries(),
13    deliv_q = [] :: q_entries()
14 }).

```

Listing 1: Node record

Because we have a single ring and not a ring of rings, we need to find an alternative way to store what users and messages are in a chat(node). For this we simply used a record for the node which included a list of

User records as well as a list of Message records.

```
1 % Record for the users
2 -record(user, {
3     name :: string(),
4     pid :: pid()
5 }).
6
7 % Record for the messages
8 -record(message, {
9     user :: user,
10    text :: string()
11 }).
```

Listing 2: User and message record

As seen above, we have a record for a user, which holds the name of the user as well as a pid. The idea is that a user sends a message to the node in charge of the chat, which then sends the message to all users of a chat. This is to simulate that the chat simply shows the message in the chat itself.

The message record simply holds information about the author of the message as well as the message itself.

Having a single ring allows us to acquire all information we need with a single run through the ring. We can simply use the successors to form a kind of "linked list". We wrote a piece of code that would go through the ring and stop when it ended up at the node where it started. If we were to have a ring with multiple rings inside each of them, this process would be much more difficult and computational expensive.

## 2.1 User Interface

The user interface, later referred to as UI, was implemented as a CLI (Command Line Interface). This was chosen because it made it possible to implement the different functionalities without having to introduce a new language or tool. Because it is a CLI it was possible to implement it in Erlang, the functions mainly used is `io:get_line` and `io:format`. The code is also split up into a number of loops, this is done such that it is possible to repeat some of the prints without having to repeat the whole code. An example is when the user writes something the UI does not recognize the user should get the chance to write something else without the UI repeating unnecessary things.

When the UI starts up a welcoming screen is printed where the different options are listed. This start screen can be seen in Figure 3. There are 6 different options. The first is listing the different groups in the system. Here a message is sent around the ring asking each node to sent back it's name and then sending the request along to its successor. The UI then prints out all the names it got from the ring.

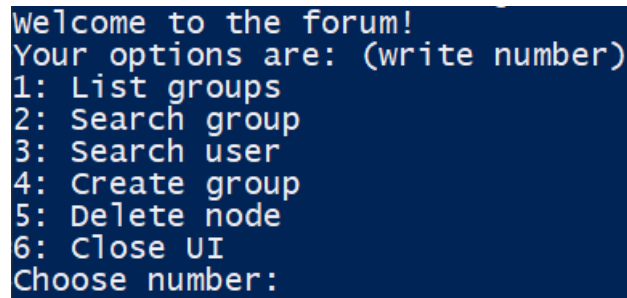
The second option is searching for a specific group name, again it collects all the names of the nodes/groups in the ring and then it checks if the name searched for is in the list of names it got from the ring.

Option 3 works the same way though it is the different nodes' list of users that it requests.

Option 4 is creating a group where you give the UI a group name and it then tells the ring that a new node is joining the ring with the specified name.

The option 5 is more for testing that the ring is able to stabilise if a node in the ring shuts down. It tells the specified pid to exit.

All the options trigger new prints which give the user different possibilities. For example when listing



```
Welcome to the forum!  
Your options are: (write number)  
1: List groups  
2: Search group  
3: Search user  
4: Create group  
5: Delete node  
6: Close UI  
Choose number:
```

Figure 3: Start screen of the UI

groups you will have the option to connect to one of them, or go back to the start screen. It is the same in option 2 when the group you have searched for is found. When connecting to a group it asks if the user wants to see the users already connected to the group. If the user says yes the UI asks the group node for its list of users. After this the user must specify a username. Then it takes the messages stored in the group node and prints them. This means that a user can see the messages written in the group chat before they connected to it. At this point the UI spawns two different loops, one which takes care of getting the messages that are done being processed by the total ordering algorithm and printing them in the correct order. The second loop waits for the user to write something and then sending that message to the node triggering the total ordering algorithm. The original UI process runs a loop that takes care of all the messages sent by the total ordering algorithm.

It is possible to quit the group chat and go back to the start screen where the user again can chose one of the options. The way to exit the UI is choosing option 6 when on the start screen. This option kills the UI process, and since the other processes that was spawned was also linked they are also killed.

The UI is simple but it has the functionalities that were required for this project and the implementation process was simple as it was written only in Erlang.

## 2.2 Algorithms

As mentioned earlier, a requirement for the project was causal ordering of messages. To implement this, we looked at different algorithms and found that the total ordering algorithm (Algorithm 6.5 in the Red Book), was the easiest to implement and required less messaging overhead between nodes than a causal ordering algorithm. This total ordering algorithm enforces both causal ordering as well as total ordering.

The total ordering algorithm requires the sender to send its message together with a unique tag and a time, to everyone who should receive the message. The receivers then store the message while replying back with their time. The sender then calculates the max of the receivers' times and send out to all of them what the max time was, and they update the time stored with the message.

For this to work with the implementation of the UI and chord, we had to change the algorithm, such that the sender sends its message, tag and time to the group(node), which sends the message, tag and, time to all users in the chat(including the sender) and asks all users for their time. The node receives all the times and calculates max, and sends to all users what the final time became. They all update their lists of waiting messages, such that the new message can be displayed to the user after a short waiting time. The function responsible for going through the waiting finished messages has a waiting time. This way if a message should appear that actually had an earlier time, than messages already in the list, then that message will be

added to the list and printed in the UI in the correct order. A sequence diagram of the messages sent in the total ordering algorithm can be seen in Figure 4. As we assume to have a reliable network and unbounded bandwidth, no message will disappear.

This means that instead of the original sender taking care of the max time, the group(node) does this.

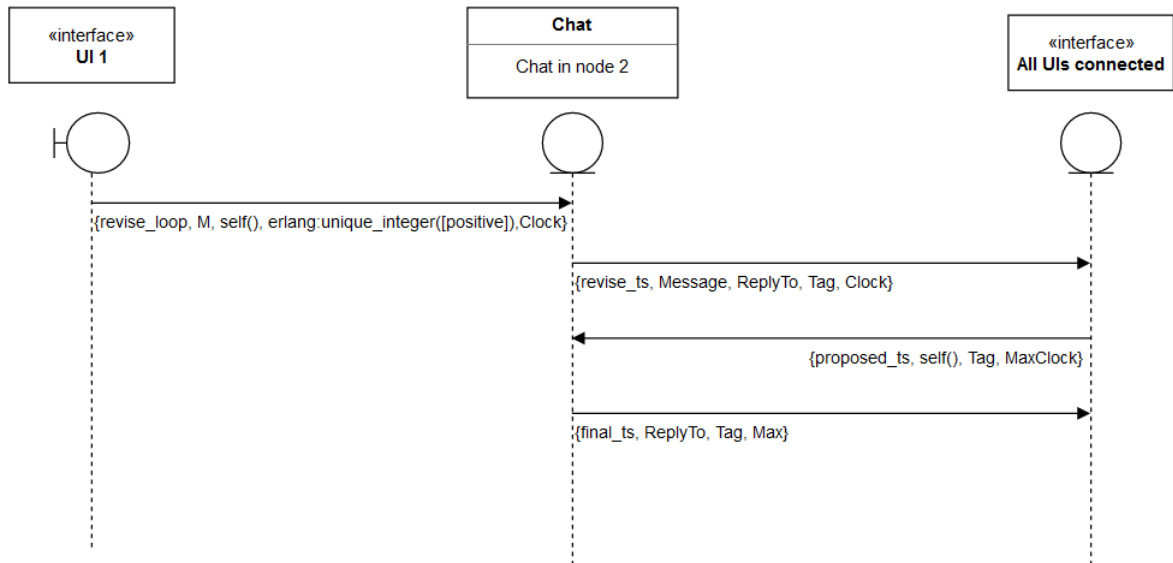


Figure 4: Sequence diagram of total ordering messages being send. All UIs connected include UI 1, such that revise\_ts and final\_ts are also sent to UI 1.

## 2.3 Challenges

During the implementation of the chat, we faced different challenges. For the architecture we had to decided how we should use Chord for the chat system. Should each node in the ring be a user or a chat? If a node was a user we would end up having rings in a ring, such that each ring would be a chat. This seemed to create a lot of difficulties for traversing rings, and for finding the chats and users, which the user could search for. Choosing to have one ring, where each node resembled a chat, made things more simple as we would only have a single ring, which would be easy to traverse when acquiring information, and would not bring difficulties when adding a new user, as they would not need to become a node in the ring, and we would not have to figure out how to add the user to the correct ring in the ring. This design choice also as the advantage that it does not affect the ring at all if a user crashes, there is no need for stabilising. The code also has timeouts when waiting for responses from the users meaning that it will never wait forever if a user is down.

When making Chord failure tolerant, we had to add a list of successors to all nodes, such that if their successor shut down, the node would just change its successor to the next in the list. A challenge for this, was handling the successor list when a new node entered the ring. When should the list be made? Did stabilize need to run first? As we figured out, we had to get a correct successor before making the list, as we had to get the successor's successors to avoid traversing the ring, because if nodes crashed during this search, what would happen to the list; which node would have which as its successor and predecessor; ect.? Another challenge which arose from making Chord failure tolerant was inconsistencies in the view of the system because of the time out and interval in stabilize. We had to figure out how to set the time out and the interval, such that stabilize was running often enough, but still allowing messages to be send and acted on by other processes, such that nodes did not disappear in the view of the system or were thought to have died.

A challenge for implementing the total ordering, was figuring out which process had which responsibility, because we have the group in the middle of message passing between users. We e.g. had to find out if the group should be considered a receiver of the sender, or as we chose to do, have the group act as a multi-caster for the sender.

### 3 Assumptions

We assume that the network is reliable, homogeneous, and secure. These assumptions allow us not to worry about networking as well as system insecurities, in general. This makes our prototype much easier to write.

We assume that the bandwidth is unbounded, meaning that we have unlimited data transfer speed inside of the system.

For an easier implementation, we assume a stop-fail model for failures, meaning a node will not be automatically started back up after failing. If a node fails, it will be removed from the network. This assumption allows us to not have to backup nodes and have failure resilience, as well as not to worry about nodes starting with an out of date or corrupted state.

We assume that the system is stable and that no data needs to be persistent. In other words; if a node fails, all the data in the node will be lost. As previously mentioned, a node is a chat. This means that if a chat crashes, it is simply removed from the ring of chats and the ring is stabilized again. The messages and users stored in the chat will be lost.

### 4 Correctness

The implemented system is semi-decentralized since all UIs connect the users to the same master node when using the chat system. This master node is the point of contact the UI has with the ring.

The semi-decentralized structure means that if the master node crashes then it will no longer be possible to join the ring with the before known information. The ring is still there, if any group has been created, so the users need the information of one of the other nodes in the ring, but this is not implemented in the prototype.

As the application is implemented at this time when a node crashes a new node is not started up. As mentioned before is it necessary to do something then the master node crashes to maintain the communication between the UI and the ring. One way of handling this, is when the predecessor of the master node notices that the master node is down it sends its own information to the users connected to the ring, then it must be implemented that all nodes know all users in the ring. Another way to handle this would be to make changes such that when the UI joins a group, this group is now the point of communication for the UI and the ring, instead of the master node. The UI would still know the master node and use that for communication if the group is removed. There is still a problem if both the group and the master node has crashed but that would require more than just one node crashing and therefore be safer. This is something that should be implemented at some point for a more fail-safe application, because it is a vulnerability in this implementation.

The semi-decentralized structure also means that if the master node crashes the groups that have already been created will still be up and the users in the groups will still be able to communicate with each other. So there is no single point of failure in that regard.

The chat is distributed in that the UI has a function which starts up the ring and creates the masternode. This function then prints a tuple that is needed for users to connect to the ring. The tuple has the information of the masternode (its pid and on what Erlang node it is on). The users can then be on different Erlang nodes and communicate with the same ring and writing messages to each other.

The structure delivers availability for the group chats in the ring and the self stabilisation made by the chord code makes sure that the availability of the ring continues even when a node crashes.

When a user sends a message all the other users in that group chat will receive the message. Though there is some delay because of the implemented total ordering algorithm, but the messages will be delivered at some point and be in the same order for all users ensuring consistency.

The UI is a CLI and as talked about previously it uses `io:format`, so print statements, when showing information to the user. It seems that these print statements can be a little inconsistent at times. A bug is e.g when printing all the groups in the list it sometimes does not print a group, but if you know the group exists you can still connect to it. The fact that it is possible to connect to it means that it went through the ring correctly but the print is missing. It would be more stable to make a GUI but it was not deemed necessary for this project because the CLI has the functionalities that was needed.

There can be some inconsistency with the information of the groups the user sees when they connect to it. If one user joins and writes in a group after another user has listed all the different groups but before it has connected to it then the user connecting will not see what the first user has written. This is because when connecting to a node the information received when wanting to list the groups is used. It is then possible that this information is not totally up to date. A way to solve this would be to ask for the node information right before connecting the user to it, but this would mean more calls and therefore it would be slower. So in the implementation of the prototype the speed was here prioritised.

## 5 Conclusion

For our final project, we created a semi-decentralised chat-system. The system upholds the project requirements by having a UI, providing the UI functionalities of searching for groups, listing groups, searching for users, joining a group and listing users in that group. The system is fault tolerant up to a degree of 8 successive nodes failing. The messages sent in the channels are totally ordered, and thereby also causally ordered, as required by the project.

The system is also distributed and can be accessed using different Erlang nodes.

The chat-system is of course only a prototype, and still requires some work to be fully fail-safe.