DM883 – Distributed Systems

Final Project : Decentralised Chat, Spring 2021

Troels Peter Have
trhav17@student.sdu.dk

Mikkel Boger Posselt
mipos17@student.sdu.dk

Md Shihab Ullah
mdull20@student.sdu.dk

Department of Mathematics and Computer Science
Faculty of Science, University of Southern Denmark, Campus Odense

Contents

1 Introduction

For this assignment, we had a range of project options to choose from and work with. We as a group decided to design and implement a distributed chat system that supports multiple groups, and that has a decentralized or semi-decentralized structure.

In particular, the system needed to support/offer the following:

1. Listing users in a group chat and search for users based on their name.
2. Searching for groups based on their name.
3. Causal ordering of messages.
4. A message history for each group.
5. An interactive user interface

We choose to create a fully decentralized system, with a peer-to-peer architecture, using a fault-tolerant DBMS named Mnesia for all our data handling and peer replication along with gen_server for communications, as described in more details in the following sections.

2 System Architecture

As per requirement of the assignment, we have considered Peer-to-Peer (P2P) architecture which is decentralized in its nature. Just like in the previous assignment, we tried to come up with the system architecture gradually.

2.1 Work Process

Firstly, we implemented a centralized client-server model where a client can send and view messages in only one group which is basically one table in Mnesia DB.

Secondly, we tried to create multiple chat groups where a user can interact. We also tried implementing client access via erlang nodes so that multiple clients can access different chat groups from a server.

Thirdly, we made a complete centralized group chat with all required functionality, such as listing groups and its users etc along with interaction capability using shell terminal.

Finally, we converted the centralized system flow into a decentralized peer-to-peer group chat system respectively.

2.2 General Peer-to-Peer (P2P) Network

In Peer-to-Peer Network, clients and servers are not differentiated i.e. each and every node can do both request and respond for the services. Each peer has its own data along with access of its remote peer data and focus more on connectivity [1].

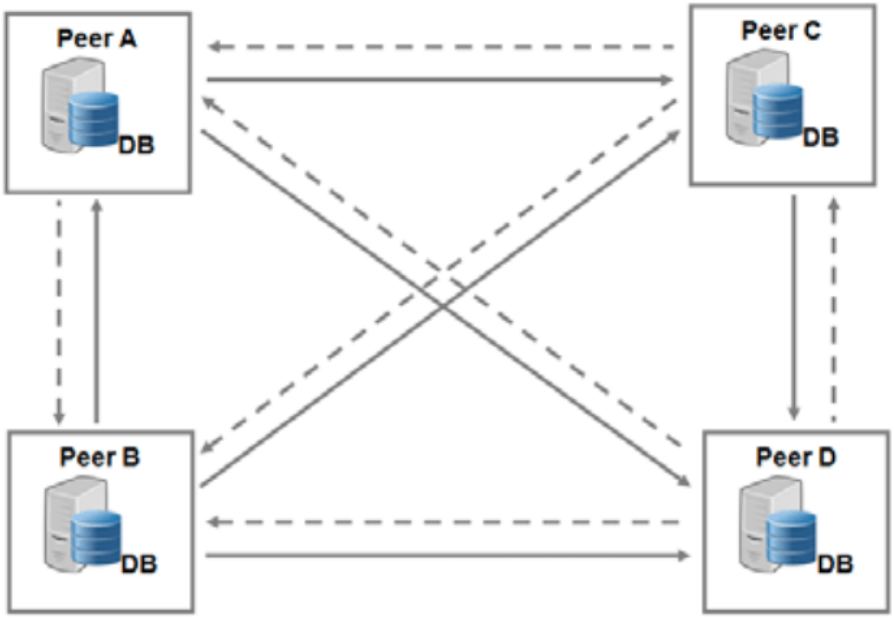

Figure 1: Decentralized Peer-to-Peer Architecture [2]

In the common client-server architecture, multiple clients will communicate with a central server, whereas, peer-to-peer (P2P) architecture consists of a decentralized network of peers. P2P networks distribute the workload between peers, and all peers contribute and consume resources within the network without the need for a centralized server. In its purest form, P2P architecture is completely decentralized. However, in application, sometimes there is a central tracking server layered on top of the P2P network to help peers find each other and manage the network. P2P network can be used for a variety of use cases, and it can easily start up another network if one is taken down, which means there is no single point of failure.

Some examples of P2P architecture are Bitcoin, BitTorrent, Skype (it used to use proprietary hybrid P2P protocol, now uses client-server model after Microsoft's acquisition) etc [3].

The network is very easy to scale up, uses available resources of peers, who normally would not contribute anything in a typical client-server architecture.

2.3 Decentralized P2P Group Chat

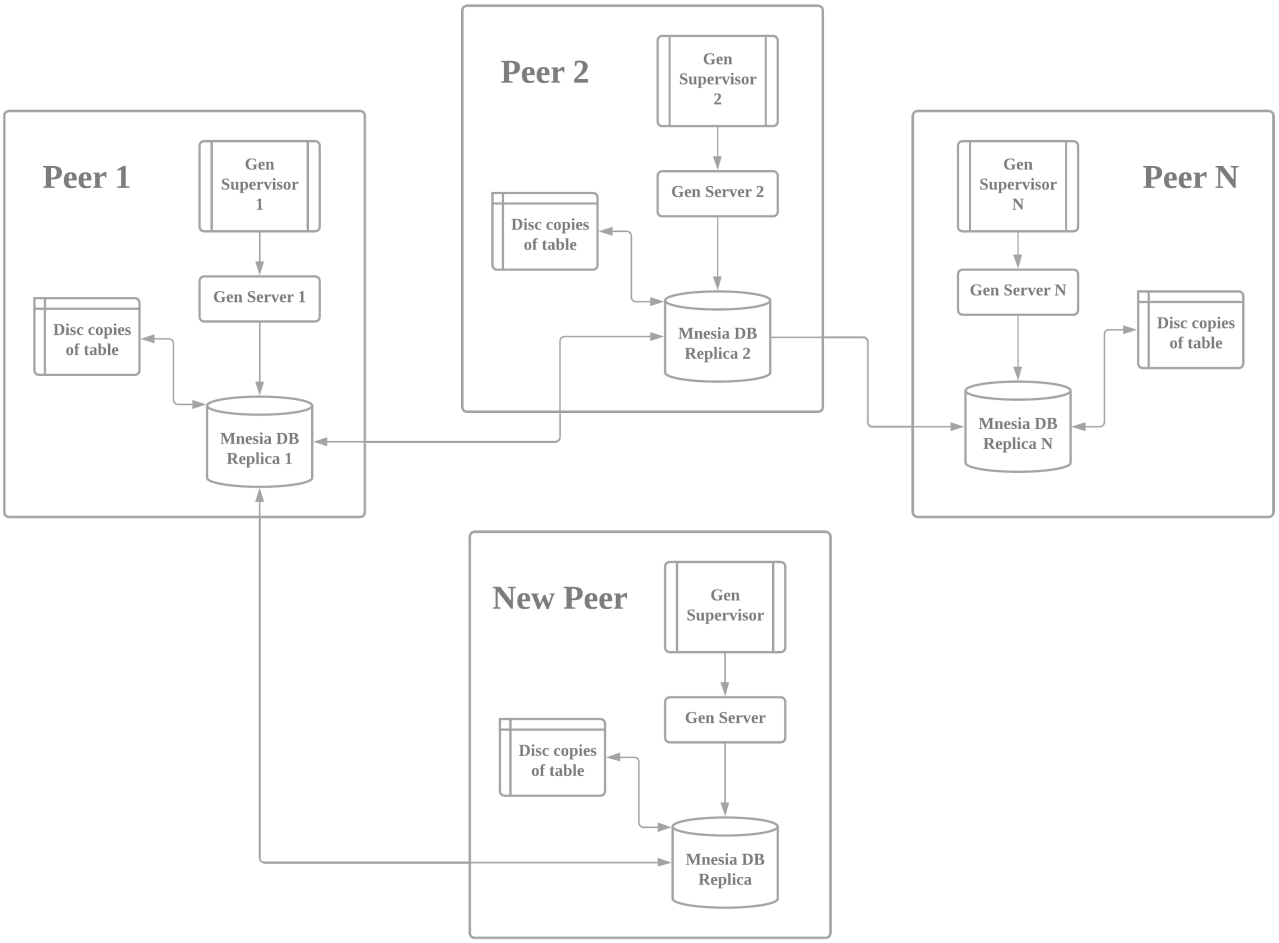

Figure 2: Implementation of Decentralized Peer-to-Peer (P2P) Group Chat System

In our decentralized P2P group chat system, each peer is comprised of three main components: Gen supervisor, Gen server and Mnesia schema.

There are some possible scenarios in our system which are mentioned as follows:

Firstly, when there is no peer existing, there is no system. To create the system, one erlang node must create the first peer by a user which start the mnesia cluster for the system by the help of gen_server via gen_supervisor. In this case, the peer will be the only group member and will be sending messages to itself via all the created groups.

Secondly, if there is already a peer existing in the system, the second peer can add itself as a peer-node using the nodename of the first peer as a reference.

Thirdly, if there are more than one peer existing in the system, a new erlang node which want to add itself in the system, can add itself as a peer-node using the node-

name of any running peer as a reference.

Fourthly, if any peer wants to leave the system, it can call mnesia:stop() in its shell or simply force kill it.

Our group chatting system is based on the idea where any peer as user can create and communicate in group with the other peers i.e open group channel.

2.3.1 Server

For implementing the server-side of our P2P network, we used the `gen_server` OTP library. The `chat_supervisor` starts a `chat_server` which creates the schema for its own DB nodes and starts the Mnesia DB.

Whenever a peer processes a call, it communicates with its own server. This server has been started and runs for only that peer. This is to make sure every user is able to exist and access the network on their own. Each server starts up its own database, and uses that for all communication, meaning no outside communication is necessary, only communication via the server to the database.

This server being a `gen_server` instance means it has some defined exported functions that can be called via exposed dedicated client functions. Synchronous requests to `handle_call/3` is used to make all the interaction to the database, as per given functions and implementation in the `db_logic` file.

The server is started up by a gen_supervisor, which is built with only a single function for this project. The only thing that is built is the function: `start_link_from_shell()`, which as it is very straight forward and starts up the gen_supervisor in the erlang shell.

This function is accessed, like all other functions, in the `chat_client`, where a user can start up a supervisor, while also starting up Mnesia locally. When the call to `chat_client:start_link()` is made, the user has started up a server, through the supervisor and is ready to communicate with it's server, which in turn. talks. to its own database.

2.3.2 Database

The decentralization of our P2P group chat system is largely based on Mnesia's table replication capabilities without incorporating master node tables. This is in case of

any database inconsistency error, so the node does not retrieve table contents from the master node, but rather it should restart. In this way we avoid any centralized data synchronization [4].

Instead of using the default `set`-type we used the `ordered_set`-type of Mnesia table so that user messages are ordered when fetched by `viewHistory(GroupName)`. Even though the average time complexity of search, insert and delete operation for `ordered_set` type table is higher i.e. $O(n \cdot log(n))$ whereas that of `set` type table is $O(1)$, in worst cases all the mentioned operation will be $O(n)$ respectively.

For a chat system, the obvious choice is to sort based on the date stored with the message. An ordered table cannot have entries with the same value for this date. If two happen to have the same value Mnesia will simply select the latest and discard the other. Luckily the timestamp in Mnesia, is down to milliseconds, and it is therefore highly unlikely that two messages will be sent to the same group at the same time.

Whenever a peer joins the group chat cluster of Mnesia, it calls its supervisor to initiate its server which starts up Mnesia. Then all the tables, representing all the groups, in the chat system are copied to the new peer except the schema which it creates by itself.

Mnesia allows for multiple tables in the database, which means it is possible for users to create groups, which will also automatically be created in the other users' databases. Since we only need one type of record (all messages are stored similarly) it is easy to simply create a new table whenever there is a call to `createGroup`. Each table uses the following record: `-record(msg, date, name, message).`

2.4 Horizontal Scaling

2.4.1 Replication of Database

We have used full table replication methods for replicating data because it is useful if records can be deleted or lost from DB nodes on regular basis due to unexpected failure or termination [5].

A full table replication copies everything from the source to the destination, including new, updated, and existing data [5].

3 Use of OTP Libraries

3.1 RPC Module

For this project we are using RPC calls to communicate with other peers in the system. Since each client is using its own separate server and database, there are some cases where we want to make a call at another client's server. This happens whenever we send a message. When call `chat_client:sendMessage(Group, Message)` is called, the only thing that happens locally is that the database is updated. Since we are using Mnesia and replicas, the databases of the other clients are also updated, however, the update is not displayed to them. This is where the remote call comes in.

We use an RPC call in order to call `viewHistory(Group)` of the group we sent the message to, for each user in the group. That way every client who is in the group, will get the message history printed out, so they can see the latest message, and the context of the message, within that chat room.

3.2 Gen Server and Gen Supervisor

These two modules are used in order to run the local server. It might be a bit overkill to run the supervisor, when there is only ever one server created, but it is a nice and simple way to make sure the user never communicated directly with the server. Everything the user does happens through a separate client, where that client module either communicates with the server or supervisor.

Gen Server is a good fit for this project as it can easily handle the different calls that might be made to it, and can easily communicate as a middleman between the client and the database.

Gen Supervisor on the other hand, is quite okay, since it doesn't bring an extra strain on the project and allows for the user to never intact with a server manually, and can also help with restarting the server, if something is wrong. As the server started by the supervisor is set to be permanent, it won't shut down unless you tell it to, and this means that the server and thus entire program isn't as volatile as it otherwise could have been.

3.3 Mnesia

For the database it was easy to pick out Mnesia. We had already learnt about it from our previous work on a Key-Value Database, so we knew how to properly interact with it, and we also learned of a valuable way to duplicate it and hold local version. That function is `addReplica`.

`addReplica` was a function we initially built for the Key-Value Database, but that functions just as well for this project, since we are dealing with locally stored databases. Due to the fact that every single client needs their own local database, we can simply duplicate a database already in existence, when a user joins the system. That way, it will act as a copy of the database, and all other copies (including the original) will be updated whenever there is a change to a database.

This is why Mnesia is so brilliant for this specific project. Every user needs their own database which they interact with. This way, every user, has their own means of communicating with the system, without having to connect outwards, outside of initially joining the system. All the Mnesia databases will remain connected and update each other with each message sent, thus completing the entire decentralized structure.

When starting Mnesia, a .LOG file called LATEST.LOG was created and placed in the database directory. This file is used by Mnesia to log disc based transactions. It also includes all operations which manipulate the schema itself, such as creating new tables. The format of the log can vary with different implementations of Mnesia. [6].

The log file will grow continuously and must be dumped at regular intervals. "Dumping the log file" means that Mnesia will perform all the operations listed in the log and place the records in the corresponding .DAT, .DCD and .DCL data files. The table content is placed in a .DCD file on the disc. For example, if the operation `"write record foo, 4, elvis, 6"` is listed in the log, Mnesia inserts the operation into the file foo.DCL, later when Mnesia thinks the .DCL has become to large the data is moved to the .DCD file. However, it is important to realize that the Mnesia system continues to operate during log dumps [6].

3.4 Lists, io_lib, String and Calendar

We have used various list processing functions from the "Lists" module for trivial list operation like finding the first element of the splitted nodename using `nth/2`, flatten datetime returned from the calendar module as per necessity using `flatten/1`, check-

ing if a user belongs to a group and also whether a group exists using `member/2` etc [7].

We have used standard `io_lib:format/2` and `io_lib:format/3` to provide necessary format for shell output such as server responses, chat history etc [8].

From the string library, we used `lexemes` to split the full node name and display only the user name from it and `titlecase` method to make the string accordingly [9].

We have used the Calendar OTP library to properly convert the generated erlang timestamp to human readable "YY-MM-DD HH:MM:SS" date-time format. Instead of UTC time we converted the timestamp into local time as that is the general convention of user experience in any chat [10].

4 System Quality Attributes

4.1 Scalability

Since every single user starts their own server and runs their own database locally, the scalability is practically infinite. There is no bottleneck at any point. The issue of scalability comes in the `viewHistory` function, since it will print every message in a group. A group chat that has existed for a long time, will have a lot of messages, so it will begin taking longer and longer to print all messages in the group.

In case of Mnesia, instead of using `ordered_set` when there will be millions of messages in a table, we can consider using set tables and employing an algorithm to make records ordered. For example: We can store a counter of each message records to keep track of their orders and then making the records ordered when writing them sequentially while being careful about handling the race condition [11].

However if we assume that the chat system will work more like forums, it might not be that group chats exist for long, and since there is infinite user scaling, users can talk to as many groups as they want, and only once a group becomes filled with a large number of messages, a problem arises as Mnesia can take considerable amount of time to make copies of big tables of group chat histories for each peer, even though the lookup-time in the group chat will be satisfactory [12].

4.2 Consistency

When a user calls `chat_client:addNode(Host)` as an erlang node, it gets attached to the Mnesia cluster and gets added as a peer. Since all users get both disc copies and remote access accordingly to all the groups i.e. Mnesia tables, there is consistency among all users on the service. When any peer joins again after being down, they get the updated chat histories and cluster details automatically.

In our system, we invoke a synchronous `gen_server:call` which makes consistency preserved while sending a request and waiting until a reply arrives or a time-out occurs [13].
Our system has been backed by strong consistency because Mnesia is an ACID in-memory DBMS. We did not use any dirty functions to read or write into our DB nodes and further used `disk_copies` to ensure it can be repaired automatically. If we had used dirty functions provided by the Mnesia library, we could have increased the read-

write performance but it would have created consistency issue as there would be no locking, local transaction storage or commit protocols involved.[14].

4.3 Availability

Whether a replicated node is dead, alive or just disconnected because of a network failure, Mnesia considers and handles it. Otherwise, we had to create some kind of distributed transaction algorithm because connections might fail and the state of the request in the remote node can be unknown[15].
At startup Mnesia tries to connect with the other nodes and if that succeeds it loads its tables from them. If the other nodes are down, it looks for marks in its local transaction log in order to determine if it has a consistent replica of its tables. The node that was shut down last has mnesia_down's from all the other nodes. This means that it safely can load its tables with casual ordering of the messages [16]. The database is non-volatile so the data does not get destroyed after complete shutdown, rather it is saved in the disc. If there is at least one peer alive, it can act as both server and client and can interact with the system and update it.

4.4 Data recoverability

As there are always disc_copies of group chats available and distributed among peers due to replication, unless every peer leaves the system by deleting their own disk_copies, any peer can be used as recovery db_nodes by other peer by just adding themselves in the cluster. Data will be unrecoverable if the Mnesia cluster is deleted from the system.

4.5 Extensibility

We can spin up as many `gen_server` instances and active DB nodes as we want. It is very easy to add a client node as a peer to our system. As it is not a client-server model, the user does not need to go through vigorous server configuration or client-side setup.

We can extend the replicated DB node capacity using fragmentation. Mnesia Fragmented Tables can be used to extend and store as much data we want by splitting a table into several manageable fragments despite table size limit. The fragmentation module named `mnesia_frag` uses linear hashing so it uses 2^n fragments which assures that records are equally distributed (more or less, obviously) between fragments.

4.6 Data transparency

Whenever a user updates a data item, the update is reflected in all the copies of the table. However, this operation should not be known to the user which is ensuring concurrent transparency. Users can smoothly chat, see lists of groups, users and synchronized chat histories, even when a replica is unavailable as long as there is still at least one available[17].

Users also do not know how many, and where, replicas exists but are given the capacity to use it whenever they want[18].

In mnesia, one does not need to state where the different tables reside, only the names of the different tables need to be specified in the program code. A program works regardless of the data location. It makes no difference whether the data resides on the local node or on a remote node. The database can be reconfigured, and tables can be moved between nodes. These operations do not affect the user programs [19].

4.7 Fault Tolerance

Mnesia tables can be replicated to several compute nodes. Write operations apply to all replicas within the context of a transaction, with the ability to update replicas that are not available through updates upon recovery.

5 Assumptions

There are a few assumptions that the stakeholders and the system users we make in this project based on how we have decided to implement it

5.1 System runs locally

However, currently we made the assumption that all peers are erlang nodes are running on the same system. This is because it was how we were able to create it. The system is built for growing in a decentralized manner where each individual node that runs, has its own server and only interacts with that, and its own database. Had we been able to construct an actual online experience, we would have done so with given time and knowledge, but so far, this is a local only experience.

The users need to use the same cookie when creating the node, in order to interact with each other and that can be done by using the call: `erl -sname alice -cookies 1234` whenever you create a node. Any peer who wants to newly join the network must know the cookies and the nodename of any existing peer.

5.2 New peer can only be added by an existed peer

On top of the local experience, all users also have to know the name of one of the hosts on the service in order to join it. Currently, the system works by a new user calling the function: `chat_client:addNode('alice@DESKTOP-BCHONID').` which connects you to the Host written within the function call. This means that you can only join the system if you know the exact node name of a user in the system.

Any user as peer can be a host however, a new user can only join as peer by knowing an existing user.

This one-time operation is needed to be add a user as peer in the system, which is there so the user can copy the database to itself locally, and that database will be updated locally from then onward

On top of that, all users in the system can be found by using the functions of the system, so even if there had been a use for host names, those would be available to you once on the system, and thus the only real hurdle here, is getting that initial knowledge of a user who is already on the system.

6 Limitations & Improvements

Like any system, there is a set of limitations which begs the improvement of our project. We tried to mention them as per necessity to get an overview of our system along with its potential.

6.1 Limitations

6.1.1 Local system

As mentioned under assumptions, we currently assume that the service is run on one single local system. This is obviously not optimal for a system that is meant to be decentralized, and thus run on individual systems with no central host.

However, there have been no information or potential information available to us, that would have allowed us to actually implement the system on an actual decentralized platform. We have looked at different sources trying to figure out how implementation on multiple systems function, but we decided to stick to the simple implementation which is acceptable given the scope of the project duration.

6.1.2 Database Inconsistency Scenarios

There are several occasions when Mnesia may detect that the network has been partitioned due to a communication failure.

One is when Mnesia already is up and running and the erlang nodes gain contact again. Then Mnesia will try to contact Mnesia on the other node to see if it also thinks that the network has been partitioned for a while. If Mnesia on both nodes has logged mnesia_down entries from each other, Mnesia generates a system event, called {inconsistent_database, running_partitioned_network, Node} which is sent to Mnesia's event handler and other possible subscribers. The default event handler reports an error to the error logger.

Another occasion when Mnesia may detect that the network has been partitioned due to a communication failure, is at start-up. If Mnesia detects that both the local node and another node received mnesia_down from each other it generates an {inconsistent_database, running_partitioned_network, Node} system event and acts as described above [20].

6.2 Improvements

6.2.1 Joining/Leaving a group

While working on the project we played with the idea of users having to join a group before actually being able to send messages, but we decided to make our group chat system without restricted access meaning all groups are public or open for any peer joining the system and can send text messages in any group they want to.

We could have still send a pretty basic join message, you just need to write it yourself, meaning the functionality of that feature is questionable.

There is one more aspect that is interesting to discuss, and that is the leave functionality. Currently there is no way for a user to leave a group, meaning once you're in, you're in for good. This is obviously not optimal as well, as you can quickly get flooded with messages from many different chats, if you keep on joining different groups. Thus, if a user is to leave, we can simply make a leaveGroup(Group) function with `del_table_copy(Tab :: table(), N :: node())` which is needed to be invoke from the erlang shell of that respective peer [21]. Even though joining/leaving group chat functionalities was out of project scope, one of the main reason for not keeping them as dedicated implementation because there was no authentication scheme on the groups anyways making it a sort of useless step.

6.2.2 Limited viewHistory

Currently whenever a message is sent to a group, we are printing the entire history of that group to everyone that is in the group which is not efficient as in worst case the lookup time complexity can be $O(n)$. We had an idea of how to deal with that, which was to update the function `sendMessage`, so when you send a message, only a part of the history is printed. This could be the latest 10 or 5 messages in the group because usually the most recent messages are needed in order to understand the context of the message.

If we can get the length of the history and count out the messages, from the newest to the oldest, then it should be possible, to only save the latest 10 messages, and print those out to each user. This would make the program more functional for more long term usage.

6.2.3 Private Messages

For this project we also looked into the idea of private messages. In this aspect we were considering making an alternate version of `sendMessage`, where it would send to a user instead of a group. This just proved to be too far from the original intentions of the project, and so we decided not to go through with it.

It is however something that might be worth looking into if the project was developed further, since an option for individual communication might be very useful for users. On top of that, the idea of private communication can also be extended to private groups, meaning groups, where there is a password, in order to join.

Both of these options could definitely be interesting for a chat service, and one that would extend the options of the users of the service.

The way to implement it, is quite clear as well. For private messages, this would simply be to send a message to another user, in the same manner it is currently possible to send to all users in a group. This would then not be stored in the database, as it would be private, but would also mean that there is no message history of those private messages.

For private groups, this could be done by having the user creating the group, specify that it is a private group, and then have to enter a password. Then, you would need to join a private group before being able to send messages in it. This would be by calling a join function and then entering the password.

Both of these extensions are very interesting, but are outside of our scope. However, if we had more time, we would have looked into it, as it gives a more complete experience to the user.

6.2.4 Handling DB Inconsistency scenarios

Currently, if any database inconsistency occurs for a peer, we manually stop and start Mnesia to fix it. Sometimes restarting the whole system was a very quick fix which is not practical at all. However, there are some graceful way to mitigate related circumstances.

If the application detects that there has been a communication failure which may have caused an inconsistent database, it may use the function `mnesia:set_master_nodes(Tab, Nodes)` to pinpoint from which nodes each table may be loaded.

At start-up Mnesia's normal table load algorithm will be bypassed and the table will be loaded from one of the master nodes defined for the table, regardless of potential `mnesia_down` entries in the log. The Nodes may only contain nodes where the table has a replica and if it is empty, the master node recovery mechanism for the particular table will be reset and the normal load mechanism will be used on the next restart.

The function `mnesia:set_master_nodes(Nodes)` sets master nodes for all tables. For each table it will determine its replica nodes and invoke `mnesia:set_master_nodes(Tab, TabNodes)` with those replica nodes that are included in the Nodes list (i.e. TabNodes is the intersection of Nodes and the replica nodes of the table). If the intersection is empty the master node recovery mechanism for the particular table will be reset and the normal load mechanism will be used at next restart.

The functions `mnesia:system_info(master_node_tables) and mnesia:table_info(Tab, master_nodes)` may be used to obtain information about the potential master nodes.

Determining which data to keep after communication failure is outside the scope of Mnesia [20].

6.2.5 User authentication

As the entire network accessibility is not controlled by any super-peer, unless its not an anonymous and open group chat, there can be a master node introduced to keep peer information such as process ID, host name or IP address, username and password which can be used to create flow of user authentication. As the network grows, sharding of chat histories and user information should be implemented rather than replication strategy to avoid performance degradation. Many P2P systems have practiced similar semi-decentralized approach in system implementation.

References

[1] https://www.geeksforgeeks.org/difference-between-client-server-and-peer-to-peer-n

[2] https://www.researchgate.net/figure/Decentralized-Peer-to-Peer-Architecture-Moreo
fig15_330485258.

[3] https://student.cs.uwaterloo.ca/~cs446/1171/Arch_Design_Activity/
Peer2Peer.pdf.

[4] https://stackoverflow.com/questions/3573404/what-is-the-significance-of-a-mnesia-

[5] https://www.stitchdata.com/resources/data-replication/.

[6] http://erlang.org/documentation/doc-5.6.3/lib/mnesia-4.4.3/doc/html/
Mnesia_chap7.html.

[7] https://erlang.org/doc/man/lists.html.

[8] https://erlang.org/doc/man/io_lib.html.

[9] https://erlang.org/doc/man/string.html.

[10] https://erlang.org/doc/man/calendar.html.

[11] https://stackoverflow.com/questions/34176132/erlang-is-there-anything-in-mnesia-s

[12] https://www.slideshare.net/KaiZhou5/have-fun-with-erlang-1.

[13] https://erlang.org/doc/man/gen_server.html.

[14] https://www.erlang-factory.com/upload/presentations/286/
MnesiaEFLondon2010.pdf.

[15] https://stackoverflow.com/questions/62072380/high-availability-in-erlang.

[16] http://erlang.2086793.n4.nabble.com/About-mnesia-in-multi-nodes-td2119396.
html.

[17] https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_
distribution_transparency.htm.

[18] https://www.ques10.com/p/24028/explain-the-different-forms-of-transparencies-in-d

[19] http://erlang.org/doc/apps/mnesia/Mnesia_chap5.html#id75194.

[20] https://stackoverflow.com/questions/8654053/rabbitmq-cluster-is-not-reconnecting-a

[21] https://erlang.org/doc/man/mnesia.html#del_table_copy-2.