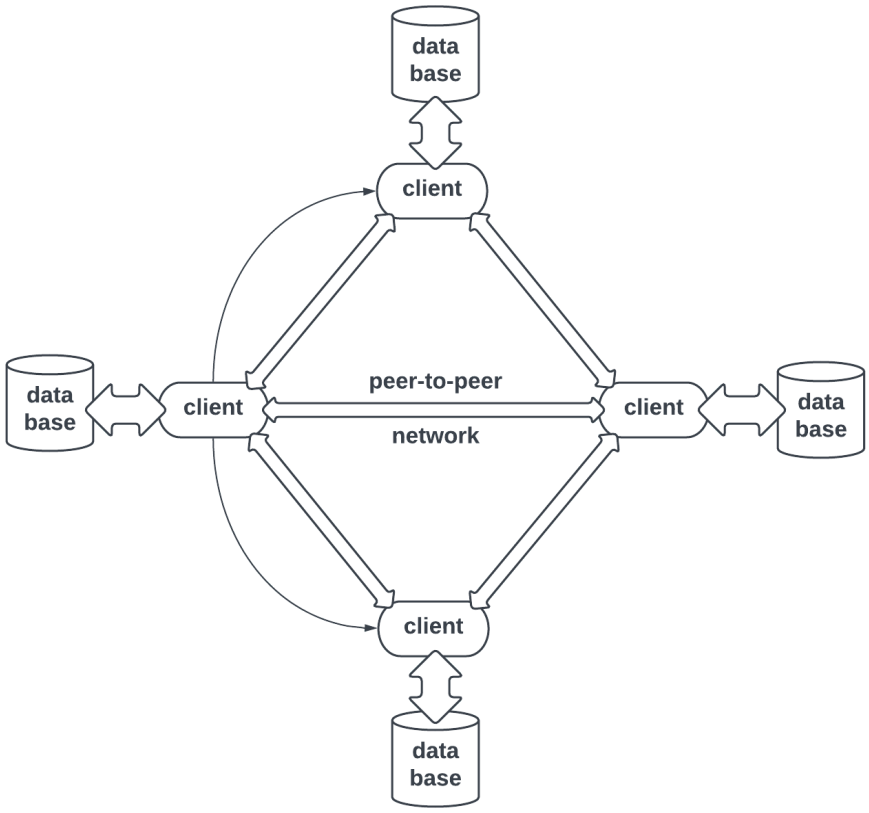
# Distributed Chat Application

Rohit Awate, Debajyoti Chakraborty, Rohan Devasthale
Github: https://github.com/RohitAwate/DecentralizedChat

## I. INTRODUCTION

We have built a distributed chat application which allows the users to communicate with each other and in groups on the platform in real time. It is a completely decentralized application in which clients use a peer-to-peer network for communication among themselves and in groups. It is built entirely in Java, leveraging its RMI technology and uses the PAXOS algorithm as backbone to achieve consensus and fault tolerance. The distributed architecture for this application allows for scalability, availability and maintainability. Our application allows users to send text messages and files to other users and chat in a group. It also supports multi-group functionality where one user can be part of multiple groups and chat with members of that group.

## II. ARCHITECTURE DIAGRAM



The diagram above depicts the architecture for our application. Since it's a decentralized application, there are no servers and clients communicate with themselves using the peer-to-peer network. Messages sent by every client are stored on disk in binary form. Similarly, all messages of a particular group chat are also stored. The clients have the ability to go offline for a while and again be active in a particular group. Our system ensures that the members of a particular group are in sync at all times even after they are offline for a while.

## III. APPLICATION WORKFLOW

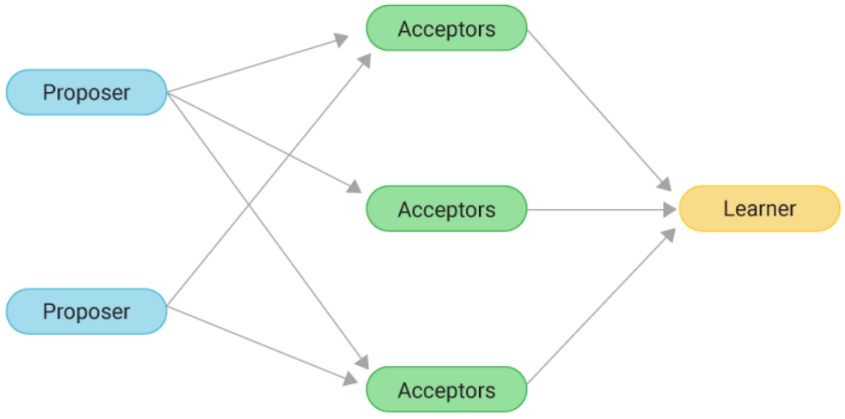
- User A starts the app. They have an option to join an existing group or to start a new group chat. Let's say they create a group with a unique name, say "BSDS_Chat".
- User B starts the app and wants to be part of the group "BSDS_Chat". We assume that they know the IP address of A and the name of the group they want to join. B requests A to let them join the group. Each user maintains a table with the names of the groups and the members within each of them.
    - A will add B's IP address to its table against "BSDS_Chat" and send an OK response back.
    - After seeing OK, B will add the group to its table along with A's IP.
- Now, say a third user, C, starts the app and wishes to join "BSDS_Chat". They may connect to either A or B to do so, depending on whose IP address they know. Let's say they connect via B.
    - C will send B a request saying that it wishes to join "BSDS_Chat".
    - B runs PAXOS with A to gain consensus in letting C join the group chat. C can join the chat only if consensus is reached. If PAXOS fails for some reason, B tries again to achieve consensus.
    - After that, B will return an OK message back to C which will also contain its own table. This will allow C to communicate with the rest of the group.
- In terms of messages, say A sends a message to the group. It will utilize its table to send a message via PAXOS.
- All messages have a unique ID and are stored in a database (disk in this case) on the client's machine.
- Clients may also go offline anytime. When they come back online, they again send a special message 'SYNC_UP' to one of the members in the group. Again, PAXOS is run by that member already in the group to achieve consensus and then the messages are sent to the client who wants to come online so that they are synced up with the group's messages.

# IV. KEY ALGORITHMS

## 1. Peer-to-Peer Network

A peer-to-peer (P2P) network is a type of decentralized network where each computer, or node, on the network acts as both a client and a server. In a P2P network, nodes can communicate and share resources, such as files or processing power, with each other without the need for a centralized server. In a P2P network, nodes can connect directly with each other or indirectly through other nodes. This means that each node can both request and provide resources, making the network more efficient and fault-tolerant. P2P networks are used in a variety of applications, such as file sharing, instant messaging, and online gaming. One of the most well-known examples of a P2P network is BitTorrent, which is used for sharing large files across the internet. Moreover, P2P networks can also be used for more complex applications, such as content delivery, distributed computing, and even blockchain technology. There are three main types of P2P networks, including unstructured, structured, and hybrid P2P networks. We have implemented an unstructured peer-to-peer network for our application. In this type of network, nodes are connected in a random fashion, without any specific organization or hierarchy. For our application, nodes are connected on the go that is whenever a client joins a chat, it gets connected to the network.

## 2. PAXOS




The PAXOS protocol is a consensus algorithm used in distributed systems to ensure that multiple nodes agree on a value, even if some nodes fail or messages are lost. The PAXOS protocol is based on a quorum of nodes that communicate with each other in order to reach agreement on a value. Each node has a unique

identifier and can be in one of three states: proposer, acceptor, or learner. We had already implemented PAXOS for project 4. Thus, we decided to leverage it and build our application on top of it. Below are the three main phases of this protocol:

- **Phase 1 - Prepare Phase:** In this phase, a proposer node proposes a value to the acceptors. The proposer sends a "prepare" message to all acceptors with a proposal number (n) that is greater than any proposal number it has used before. The acceptors respond with a "promise" message that contains the highest proposal number they have seen so far and the value associated with that proposal number. If an acceptor has already accepted a value, it includes that value in its promise message.
- **Phase 2 - Accept Phase:** If the proposer receives promises from a quorum of acceptors, it sends an "accept" message to all acceptors with the proposal number (n) and the value (v) that received the most promises in the prepare phase. The acceptors accept the value if it has not already accepted a higher-numbered proposal, and send an "accepted" message back to the proposer and learners.
- **Phase 3 - Learn Phase:** Once a proposer receives acceptance messages from a quorum of acceptors, it sends a "learn" message to all learners with the proposal number (n) and the value (v). The learners record the proposal number and value as the agreed value for that slot.

The Paxos protocol ensures that if a proposal is accepted by a quorum of acceptors, then it is guaranteed that no other proposal with a higher number will be accepted in the future. The protocol also ensures safety, meaning that once a value has been chosen, it will remain chosen even if messages are lost or nodes fail. However, one of the challenges of the PAXOS protocol is that it can be difficult to implement correctly, and there are many variations of the algorithm that have been proposed to make it more efficient and easier to understand.

3. **Distributed Mutual Exclusion and RMI**

We have leveraged Java's RMI technology to build our application. Further we use distributed mutual exclusion while saving the messages for each client. We use Java's 'synchronized' key-word as well to achieve this task. This key-word handles locks behind the scenes. RMI is based on a client-server model, where the server provides a set of services that the client can access remotely. The client invokes methods on objects that are hosted by the server, and the server processes the requests and sends back the results to the client.The server provides the services that the client can access, and the client accesses those services remotely.

Since we have a peer-to-peer network, a single node acts as both client and server. To create a node, we have defined multiple remote interfaces that define the methods that the server will provide. Remote interfaces extend the java.rmi.Remote interface, and each method in a remote interface must declare a java.rmi.RemoteException, which is thrown if there is a problem with the remote method call. Further, RMI handles parallel execution behind the scenes which also allows support for multithreading.

4. **Group Communication**

   Group communication refers to the exchange of messages among a group of nodes in a distributed system. In this type of communication, a node can send a message to multiple other nodes in the group simultaneously. Group communication is an essential building block for many distributed applications, such as replicated systems, fault-tolerant systems, and distributed databases. In message-oriented group communication, messages are sent and received among the nodes in the group using a message-passing mechanism. The most common message-oriented group communication protocol is the multicast protocol. In this protocol, a node can send a message to a group of nodes, and the message is delivered to all the nodes in the group. The advantage of the multicast protocol is that it can reduce network traffic since a message is only sent once and then delivered to all the interested recipients. We have used Java's multicast socket library for group communication for our distributed chat application. Further, group communication protocols can also support different communication semantics, such as causal ordering, total ordering, and FIFO ordering. These semantics define the order in which messages are delivered to the processes in the group, which is important for ensuring consistency and correctness in distributed systems.

5. **Time and Clock**

   In distributed systems, time and clocks play a critical role in maintaining consistency and ordering of events. Time is the measurement of duration between events or the sequence of events occurring. In a distributed system, time is not always well-defined because of the lack of a central clock or reference time. Different machines or nodes may have slightly different clocks that drift over time, and this can cause inconsistencies in the system. For our application, we record the timestamps while saving the messages on the disk. This helps us to maintain the causal ordering of messages which we use to sync-up a particular user after he comes back online. Since we currently run our application on localhost, the timestamp generated by the local machine is enough for our distributed chat application.

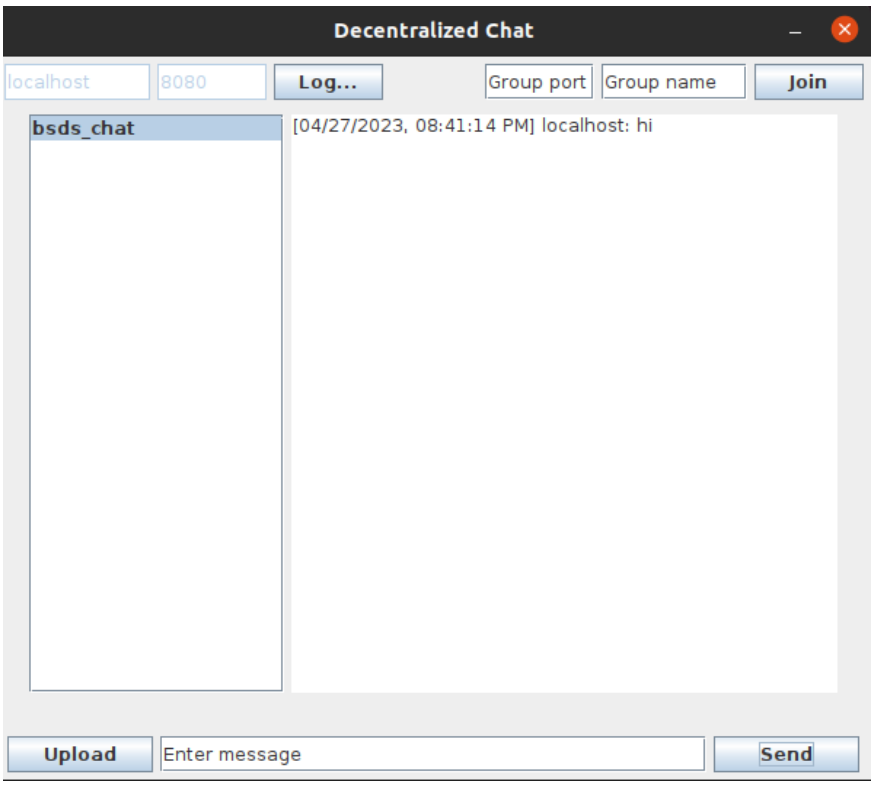
## V. IMPLEMENTATION AND TESTING

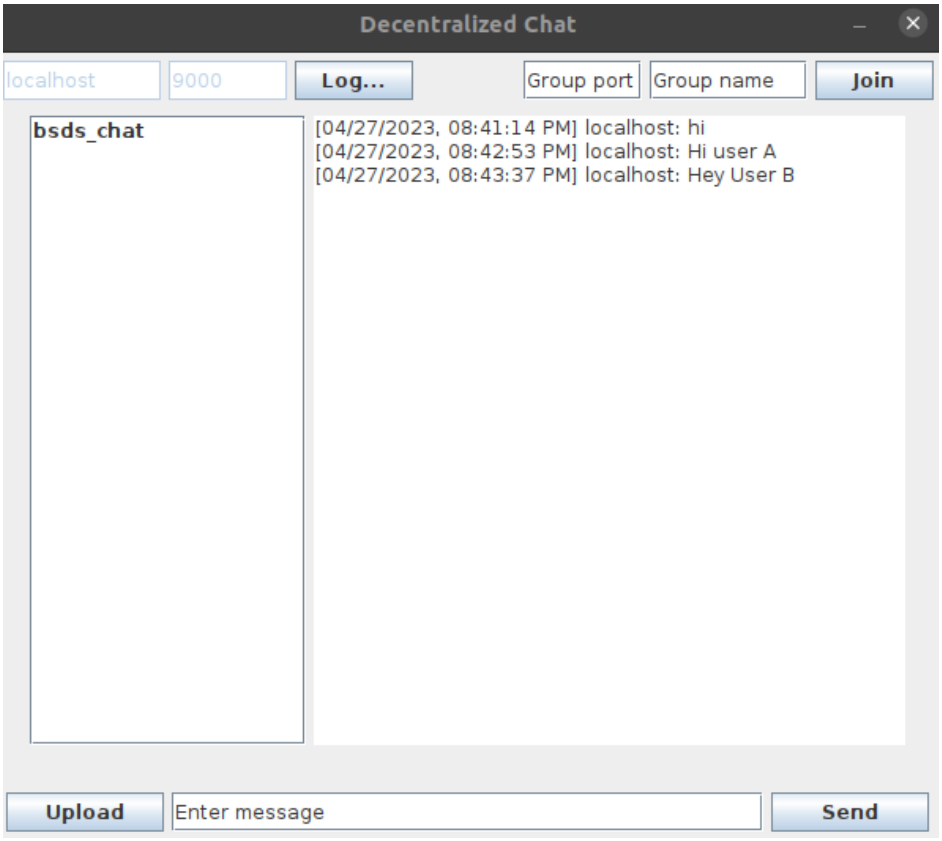
We have implemented our project primarily in Java. Further, we are saving the messages of each user on disk using Java Serialization in binary form. Our application has a simple but robust UI for interacting with users. Below is the list of libraries that we have used for the implementation of this application:

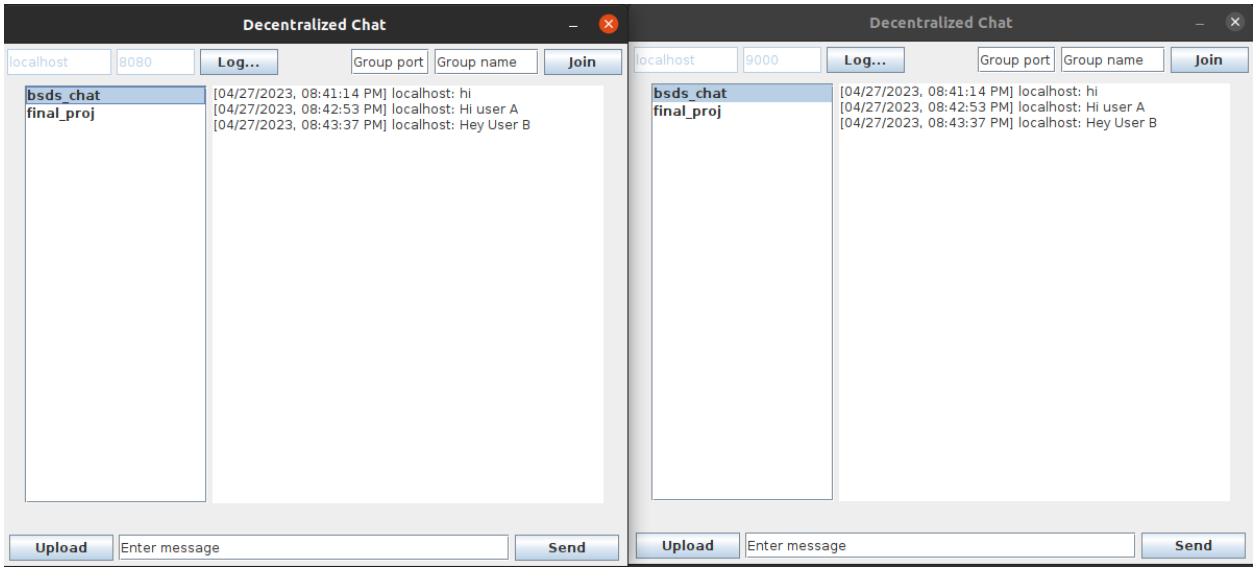
- **Group Communication**: java.net
- **User Interface**: Simple but powerful UI using javax.swing to display user chat
- **RMI**: java.rmi
- **Database:** Local disk of the client using java.io.Serializable in binary format

We implemented the PAXOS protocol that we did for project 4. Apart from the above mentioned major libraries, we also used multiple helper libraries as well. One such example is java.util library for data structures.
We tested our application on localhost for various scenarios. It involved testing for working of PAXOS protocol, group chat creation, adding a user to existing chat, sending messages and synching the user up with messages when it is back online. Further, we also tested fault tolerance of the system.

## VI. RESULTS






## VII. LEARNINGS AND FUTURE SCOPE

We learned a lot about practical distributed systems implementation from this project. One of the important things we learnt was the implementation of PAXOS for a peer to

peer network. Further, we also realized the certain limitations of RMI for a chat application and what are the alternatives for the same.
We believe that our implementation of a distributed chat system is simple yet effective. However, such applications in the real world are more complex and involve leveraging various technologies. Considering the time constraints, we have built a simple application which supports sending text messages and files using key features of a distributed system. Also, features like notifications, message queue, editing messages, support for emojis can also be considered as a future scope. Also, the user interface can be improved but since this was not the prime focus of the project, we have kept it simple.

## VIII. REFERENCES


- Paxos
- Java Remote Method Invocation Distributed Computing for Java
- What's a Peer-to-Peer (P2P) Network? | Computerworld
- https://people.cs.rutgers.edu/~pxk/417/notes/paxos.html
- https://lamport.azurewebsites.net/pubs/paxos-simple.pdf