Serverless peer-to-peer communication Android chat application

(This image is made using generative AI)

Ida Dyremosegaard-Hansen, 71336
Rasmus Laue Petersen, 68907
Kristoffer Borreschmidt Hansen, 71574
Jatinder Singh Rooprai, 66508

Characters: 108.592

1 Abstract

This paper explores the possibility of creating a completely serverless peer-to-peer communication network in the shape of an Android chat application. In the paper the work methodologies and the testing approach in the development of such a peer-to-peer network is showcased. The paper explains concepts such as what a peer-to-peer network is, along with concepts used in the code, including the Chord protocol, TCP, UDP, data transformation, encryption, and hashing. It continues on to go over the iterative design process for the application, which ends in a wireframe of the envisioned final product. Hereafter, the paper goes into depth with the code of the application, and what choices we made and why. Here the unit testing and known errors are showcased as well. Furthermore, potential attacks are considered and acknowledged. Thereafter, it is discussed what the strengths and weaknesses of the application are, what can be improved upon, and what was learned during the development of the application. Finally, it has concluded that while it is a heavy assignment building a peer-to-peer network it is a great way to learn about it in depth and that an efficient serverless peer-to-peer chat application is achievable.

2 Introduction

As a group of individuals passionate about computer science, we are eager to explore its future and the role we might play in shaping it. Our interests align in the potential of peer-to-peer (P2P) technology, which enables user and device connectivity without relying on a central coordinating entity. We envision P2P to enhance privacy by reducing the need for constant user monitoring.

The risks of centralized systems are evident. Corporations may sell user data (MacMillan, 2018), and even the most secure organizations are vulnerable to cyber threats (Newman, 2018). As we consider the future of P2P, we believe it has the potential to address these concerns in a profound way.

To better understand P2Ps potential, we decided to build a P2P chat application. Chat applications are ubiquitous, and if we can successfully implement P2P in this context, it could pave the way for other P2P applications. However, this would require the P2P network to scale exponentially to handle a large user base and constant message traffic with minimal lag.

We also chose to implement the Chord communication protocol within our P2P network, adding an extra layer of complexity to our project. By using Android for development and testing, we can leverage our familiarity with the platform to streamline the process.

Guided by these principles, we based our thesis on exploring the potential of P2P in chat applications and beyond. As we look to the future, we are excited about the possibilities of working with P2P technology and the impact it could have on privacy and security.

This has led us to the following thesis for the project, followed by research questions to aid the process of researching the thesis.

2.0.1 Thesis

How can you create a serverless peer-to-peer communication network as a chat application based on Chord protocol?

2.0.2 Research questions

In this report, we explore the following research questions that are critical for understanding and advancing the design and implementation of the peer-to-peer communication networks. These questions serve as the backbone of our investigation, guiding our explorations.

1. How can one build a peer-to-peer communication network?

2. How can a node efficiently find and connect to other nodes in a serverless peer-to-peer network?

3. What challenges arise from implementing a peer-to-peer network based on Chord protocol?

4. Which considerations can be taken to improve the communication security in a peer-to-peer network?

2.1 Scope and limitations

The first limitation we decided on was we wanted to make a peer-to-peer (P2P) network. This was because we found the concept deeply fascinating, and we hoped by exploring it we could learn a lot more about it and how it works. Thus, we also decided that we wanted to entirely focus on P2P networks and how to develop it and decided to quickly drop researching things like databases and different types of middleware so we could entirely learn and create a genuine P2P network.

To create this P2P network thereafter we decided to limit it to an Android application as that is a good way to make it mobile and allow us to frequently test it in real life conditions. For an Android application, if made native, it needs to be developed in android studio where the choice in coding language we decided to go with Java as while not as advanced as Kotlin it is a language we are more familiar with.

In our program we decided to restrict ourselves to only connecting via the local network. While making nodes outside of the local network and thus making it possible to connect over the internet would be a good feature, it was outside the scope of what we wanted to achieve with our application development.

Finally, we decided that we wanted to research end-to-end encryption in the communication. This opens the door to many cybersecurity angles, but we mainly want to restrict it to research an encryption algorithm and then just consider what other kinds of vulnerabilities our application has.

2.2 Requirements

We have determined that a serverless, peer-to-peer (P2P) network is the optimal architecture for our requirements. This design ensures that no data is stored on the network, alleviating concerns about developers mishandling user data. Instead, users establish direct, TCP connections with one another, and administrators cannot access these communications. For added security, any saved conversation data is stored solely on the users' personal devices, not on the network. We will implement this chat application as an Android app, leveraging Java for development, due to experience in developing Java applications. Critically, the application will explore end-to-end encryption in its messaging protocol, safeguarding communications from interception or alteration by malicious parties, thereby bolstering users' confidence in the privacy and integrity of their interactions.

3 Method

In this section, we outline the research methods used to investigate the key questions related to peer-to-peer communication networks. We describe both the theoretical frameworks and practical techniques applied, providing a comprehensive view of how we addressed the complexities of the study.

3.1 Work methodology

In work methodology, we will go over all the methods we use to plan the work on the application. This includes the work methodology of agile development and the tools we have used for the project.

3.1.1 Agile development

Agile development is a flexible and iterative approach to project management and software development that emphasizes rapid delivery of high-quality software. It encourages adaptive planning, evolutionary development, and continual improvement, all with a focus on customer satisfaction. This methodology is grounded in the Agile Manifesto, which values individuals, interactions, and customer collaboration over rigid processes and tools (*Manifesto for Agile Software Development*, 2017).

We have chosen agile development for its adaptability and iterative nature, which is crucial in projects where requirements may change or evolve. Agile methodologies, unlike traditional Waterfall approaches, allow for continuous adjustments throughout the development process. This flexibility is essential in responding quickly to changes and efficiently meeting project demands.

3.1.2 Kanban within the agile framework

Kanban is a strategy used within the agile framework that emphasizes visual management of work. By displaying tasks on a board divided into columns such as "To Do," "In Progress," and "Done," teams can manage workflow and limit work in progress. This method improves focus and efficiency by allowing teams to adapt to changing priorities without the confines of sprints, unlike Scrum (Anderson, 2010).

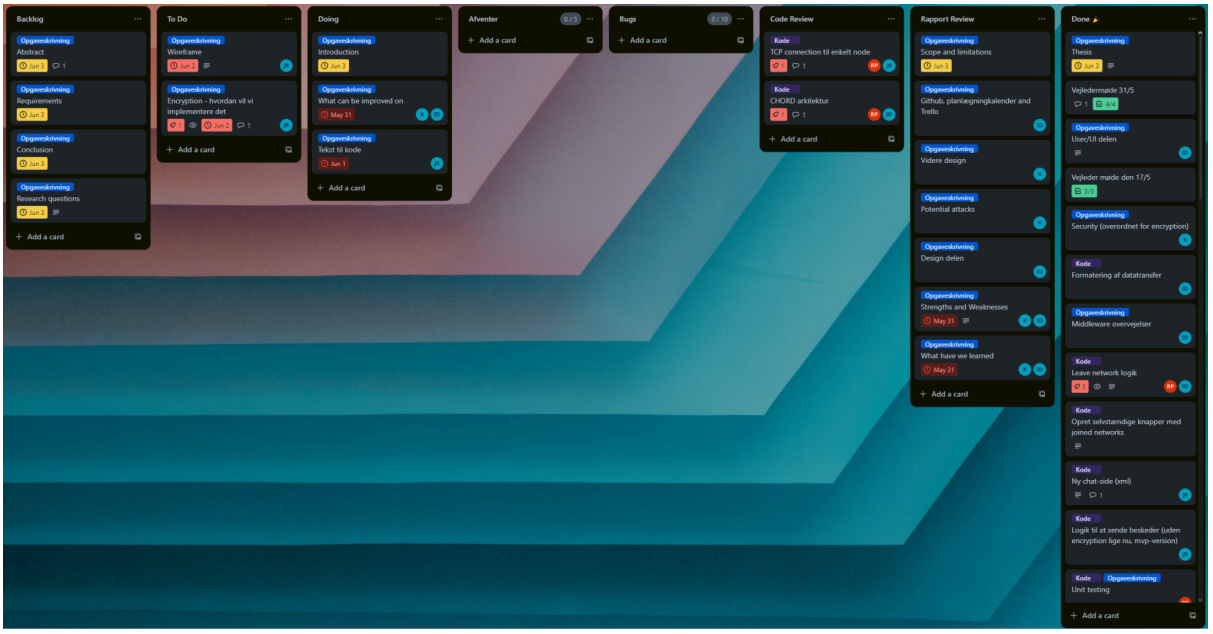
Kanban is preferred in this project due to its simplicity and continuous flow mechanism, which is better suited for environments where task volumes and priorities fluctuate. Unlike Scrum, which is structured around fixed-duration sprints, Kanban allows for ongoing adjustments based on team capacity and project needs, providing greater flexibility and responsiveness (Kniberg & Skarin, 2010).

3.1.3 Implementing Kanban in our project

In our project, we have implemented a digital Kanban board using Trello to visualize all tasks. This approach has significantly improved transparency and provided a collective understanding of our progress and priorities. By using Trello for task management, we have been able to organize tasks logically, set clear milestones and track our progress effectively.

The benefits of this setup are many. Firstly, it has allowed us to quickly identify bottlenecks so that we can adjust the division of labor effectively. By monitoring progress continuously, we can identify delays early and take immediate corrective action. This proactive approach ensures that our project stays on track and that all team members remain aligned and informed.

Overall, our use of Trello for task management has streamlined our workflow, improved communication, and enhanced our ability to manage and execute tasks effectively, leading to a more organized and productive project environment.



3.1.4 GitHub

In our project, using GitHub has offered several key advantages that enhanced our collaboration, organization, and efficiency. The robust version control system allowed us to track changes, revert to previous versions, and manage different branches without conflicts. This ensured seamless collaboration, with pull requests and code reviews maintaining code quality.

GitHubs' issue tracking system helped us manage tasks, bugs, and feature requests, keeping everyone informed about progress and priorities. The platform promoted transparency and accountability by logging every change with commit messages and author information.

3.1.5 Schedule calendar

	6. maj	7. maj	8. maj	9. maj	10. maj	11. maj	12. maj
	man.	tirs.	ons.	tors.	fre.	lør.	søn.
			Milepæl: Halvvejs med projektet				
Ida	Arbejde 9-17.30	Arbejde 10-13.30	Kan fra 12.00 af	12-19		12-00	
Kristoffer				x			
Rasmus	Arbejde 8-17	Arbejde 8-17		sommerhus	sommerhus	sommerhus	sommerhus
Jatinder		(Refinement på arbejde)	Arbejde 08-16	Arbejde 08-16			
Fælles	Statusmøde discord 17:30		Møde fysisk på RUC 12.00		Møde fysisk på RUC 10:00 Vejledermøde		

	13. maj	14. maj	15. maj	16. maj	17. maj	18. maj	19. maj
	man.	tirs.	ons.	tors.	fre.	lør.	søn.
Milepæle			Teori læses igennem		TCP/CHORD		
Ida	Arbejde 9-17.30	Arbejde 9-17.30				12-00	18-00
Kristoffer						x	x
Rasmus	Arbejde 8-17	Arbejde 8-17					
Jatinder		(Refinement på arbejde)	Arbejde 08-16	Arbejde 08-16			
Fælles	Statusmøde discord 17:30		Møde fysisk på RUC 10.00		Møde fysisk på RUC 10:30 Vejledermøde		

	20. maj	21. maj	22. maj	23. maj	24. maj	25. maj	26. maj
	man.	tirs.	ons.	tors.	fre.	lør.	søn.
Milepæle			Rasmus: Rapportskrivning af development 50 % færdig				Jatinder: Rapportskrivning og udvikling af Message, socket, tcp, udp, leave one-on-one chat, data transformation Rasmus og Ida: Chord
Ida		9.45-11.45 13.15-13.30	Arbejde 9-17.30	Arbejde 9-17.30	9.00-13.00 16.00-22.30		
Kristoffer							
Rasmus				Arbejde 8-24	Arbejde 00-16		
Jatinder		Arbejde 08-16	Arbejde 08-16	Arbejde 08-16			
Fælles	Møde fysisk på RUC 10.30		Statusmøde discord 17:30		Vejledermøde via teams		

	27. maj	28. maj	29. maj	30. maj	31. maj	1. juni	2. juni
	man.	tirs.	ons.	tors.	fre.	lør.	søn.
Milepæle		Rasmus: Leave network (Kristoffer): Potential attacks	Slutspurt	Discussion	Jatinder: Encryption Rapport færdig		
Ida	Arbejde 9-17.30		Arbejde 9-17.30				
Kristoffer							
Rasmus	Arbejde 8-17		Arbejde 8-17			Mormors 85 års føds	Mormors 85 års føds
Jatinder		Arbejde 08-16	Arbejde 08-16	Arbejde 08-16			
Fælles		Møde fysisk på RUC 10.30			Møde fysisk på RUC 10:30 Vejledermøde		

When initiating our project, it was crucial to ensure that we had scheduled regular meetings to maintain alignment and momentum. Planning ahead allowed us to set clear milestones throughout the project, optimize our teams' capacity, and establish a definitive deadline. By utilizing a detailed calendar and setting specific milestones, we ensured consistent progress toward our project goals.

Each week, we monitored our achievements against these goals, providing a clear visualization of our project schedule. This approach facilitated better communication within the team, enabling us to track responsibilities effectively. By having clear goals and a structured timeline, we could easily identify when and where assistance was needed, ensuring that team members could support each other in overcoming obstacles. This proactive strategy not only kept the project on track but also enhanced collaboration and accountability within the team.

Additionally, the regular review of our milestones allowed us to make necessary adjustments promptly, keeping the project agile and responsive to any changes or challenges. This systematic approach ensured that we maximized our teams' capacity and maintained a steady trajectory toward our final objectives, ultimately leading to the successful completion of the project within the established deadline.

3.2 Unit testing

Unit testing is a way to test the code and its outputs, and if they correlate with the expected outputs. This testing is an integral part of the process of verifying the outputs of the code, as expected. In larger code bases the way unit testing works is that instead of only testing the entire code's input and output, the code gets atomized and different parts of the architecture get its output and expected output checked (Desikan & Ramesh, 2006).

This is done in order to easily change different parts of the code, due to the different segments of code already being accounted for. This generally means new features can be implemented with much lower overhead. Generally, there are two different types of unit tests: Black Box Testing and White Box Testing (Desikan & Ramesh, 2006).

The main difference is simply that black box does not check what happens when the code is executed, and just checks if the output matches. White box testing focuses on the internal process and makes sure the code is executed the right way internally (Desikan & Ramesh, 2006).

4 Theory

In the theory section we will go over the relevant theories that are in use when constructing our application. These range from the system theories to the architecture that will be used for the application. Finally, the encryption theory will be explained to give a better understanding on how to secure the application.

4.1 Centralized vs decentralized vs distributed

To properly choose the right type for our project, it is important that we have a good understanding of the different systems. For this reason we will start by going over what the three different types are and where they are typically seen in business:

4.1.1 Centralized systems

In centralized systems, a single authority or entity maintains control over decision-making, resource allocation, and data management. Technologically, this often translates to a client-server architecture, where a central server handles requests from multiple clients (Steen & Tanenbaum, 2017, 228). The server stores data, processes requests, and controls access to resources (Steen & Tanenbaum, 2017, 228).

From a technology standpoint, centralized systems offer simplicity and efficiency in management. They often use a unified database or storage system, making data access and management (Steen & Tanenbaum, 2017, 229) straightforward. However, this centralized control introduces inherent vulnerabilities. For instance, a failure or compromise of the central server can disrupt the entire system (Stoica et al., n.d.). Moreover, scalability can be challenging, as the central server may become a performance bottleneck as the system grows.

4.1.2 Decentralized systems

Decentralized systems distribute decision-making authority across multiple nodes, eliminating the need for a central authority (Mahmoud, 2011, 2.5. Blockchain technology exemplifies decentralization, where a network of nodes collectively maintains a distributed ledger through consensus mechanisms like proof of work or proof of stake (Mahmoud, 2011, 17).

Technologically, decentralized systems leverage peer-to-peer (P2P) networks and consensus algorithms to achieve distributed consensus and maintain system integrity (Mahmoud, 2011, 66). Each node in the network stores a copy of the ledger, and changes to the ledger are propagated and validated by consensus among the nodes (Mahmoud, 2011, 68). Smart contracts, which are self-executing contracts with the terms of the agreement directly written into code, are another example of decentralized technology, often built on blockchain platforms like Ethereum (Yu & Jajodia, 2007, 383).

Decentralized systems offer benefits such as resilience, transparency, and censorship resistance. However, they introduce technological challenges related to consensus mechanisms, scalability, and governance (Yu & Jajodia, 2007, 392). For instance, achieving agreement among distributed nodes without a central authority requires sophisticated consensus algorithms, and scaling the system without compromising decentralization remains an ongoing technological challenge (Yu & Jajodia, 2007, 402).

4.1.3 Distributed systems

Distributed systems involve the physical dispersion of resources across multiple nodes or locations, regardless of whether control is centralized or decentralized. Examples include distributed databases, content delivery networks (CDNs), and cloud computing infrastructures.

Technologically, distributed systems rely on communication protocols, data replication mechanisms, and distributed algorithms to ensure fault tolerance, scalability, and load balancing (Tanenbaum & Steen, 2007, 139. For instance, distributed databases use techniques like sharding and replication to distribute data across multiple nodes while maintaining consistency and availability (Tanenbaum & Steen, 2007, 57).

Distributed systems offer advantages such as fault tolerance, scalability, and geographic distribution of resources (Tanenbaum & Steen, 2007, #137. However, they also introduce technological challenges related to communication latency, data consistency, and security (Tanenbaum & Steen, 2007, 138). Ensuring effective communication and coordination among distributed components requires advanced networking technologies and protocols (Tanenbaum & Steen, 2007, 138).

4.2 Introduction to Peer-To-Peer

In the realm of distributed computing, the indispensability of peer-to-peer systems is unparalleled, offering solutions that transcend the limitations of centralized control. At the heart of this domain lies the Chord protocol, a pioneering innovation that revolutionizes data lookup operations within peer-to-peer environments (Stoica et al., n.d.). This report meticulously examines the pivotal features and profound contributions of Chord, illuminating its role in simplifying the design of peer-to-peer systems and providing solutions for key mapping, load balancing, and adaptability to dynamic network conditions. Through a blend of theoretical analysis and practical insights, we delve into the essence of Chord, elucidating its scalability, resilience, and transformative potential in diverse applications.

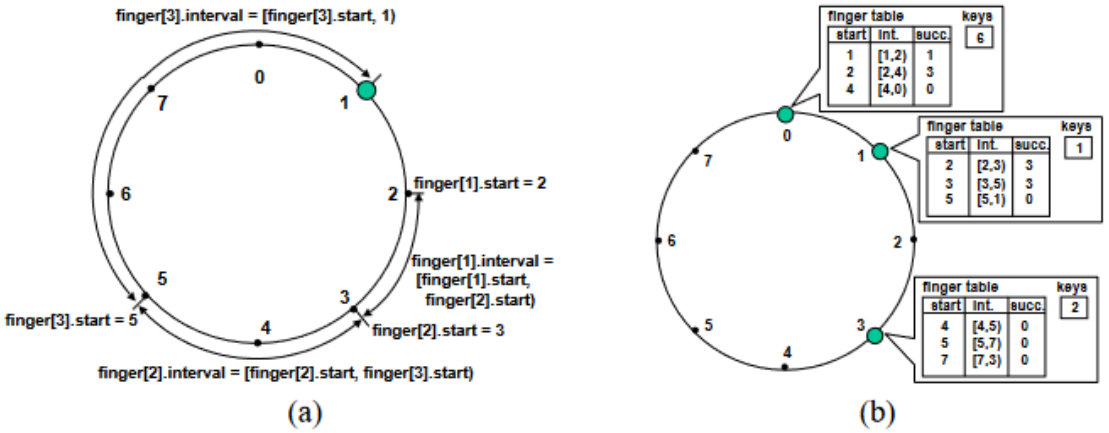
4.2.1 Chord: A Scalable Peer-to-Peer Lookup Protocol

In the intricate landscape of internet applications, the quest for efficiently locating specific data items within peer-to-peer systems emerges as a fundamental challenge. Enter Chord – a scalable peer-to-peer lookup protocol meticulously crafted to address this challenge with unparalleled efficiency. By seamlessly determining the node responsible for storing a given keys' value, Chord transcends the complexities of simultaneous node arrivals and departures, showcasing its scalability and resilience amidst dynamic network dynamics (MIT Laboratory for Computer Science, n.d.). With its streamlined approach, Chord offers support for a singular yet powerful operation: mapping a key onto a node. Through this elegant mechanism, data location becomes a seamless endeavor, enabling the association of keys with data items and their storage at the corresponding nodes (MIT Laboratory for Computer Science, n.d.). Notably, Chord's adaptability shines through as it navigates the fluid landscape of node dynamics, steadfastly answering queries amid continuous system changes. Theoretical analyses and simulations underscore Chord's scalability, showcasing logarithmic scaling of communication cost and node state with the burgeoning number of Chord nodes (Stoica et al., n.d.).

In the context of the Chord network, anchor nodes play a crucial role in maintaining stability and efficiency. Anchor nodes, often pre-designated or strategically chosen nodes within the network, serve as reference points that enhance the robustness of the Chord protocol (MIT Laboratory for Computer Science, n.d.). They assist in managing the distributed hash table (DHT) and can help coordinate the joining and leaving of other nodes (MIT Laboratory for Computer Science, n.d.). By leveraging anchor nodes, the Chord network can more effectively handle dynamic changes, reducing the overhead associated with node state updates and lookups. These anchor nodes act as stabilizing elements, ensuring that the network remains resilient and continues to function smoothly even as the number of nodes fluctuates (Steen & Tanenbaum, 2017, 105).

4.2.2 Exploring P2P Networks in Text Handling

Beyond its foundational role in distributed computing, peer-to-peer (P2P) networks emerge as potent facilitators in the realm of multimedia content handling. Within this domain, the distributed nature of P2P systems unlocks a myriad of possibilities, revolutionizing the sharing, distribution, and access to text. Through seamless file-sharing mechanisms, users transcend the confines of centralized servers, directly exchanging text files with peers across the network. Moreover, P2P networks serve as formidable allies in content distribution endeavors, efficiently disseminating large text files among multiple nodes. By harnessing the collective power of network nodes, P2P architectures alleviate the burden on individual servers, accelerating the download process and enhancing user experiences (Stoica et al., n.d.; MIT Laboratory for Computer Science, n.d.).



(MIT Laboratory for Computer Science, n.d.)

In the realm of distributed computing, the Chord protocol emerges as a cornerstone, revolutionizing data lookup operations within peer-to-peer (P2P) systems. Its architectural brilliance lies in organizing nodes in a ring topology, where each node is assigned a unique identifier. This structured overlay facilitates efficient routing and lookup, enabling nodes to swiftly locate distributed resources. Moreover, Chord's adaptability to dynamic network conditions ensures robustness, while fault-tolerance mechanisms such as redundant pointers and stabilization protocols bolster its resilience. This architectural ingenuity not only simplifies data lookup but also enhances the scalability and reliability of P2P networks, cementing Chord's pivotal role in distributed computing. Figures A and B further illustrate Chord's design, showcasing its ability to navigate complex network dynamics while maintaining efficiency and fault tolerance (MIT Laboratory for Computer Science, n.d.).

4.2.3 Maintaining Fault-Tolerance with Chord: Mechanisms and Strategies

Central to the resilience of peer-to-peer networks lies the robust fault-tolerance mechanisms embedded within the Chord protocol. Through a multifaceted approach, Chord fortifies the network against disruptions and failures, ensuring uninterrupted operations even amidst adversities (Stoica et al., n.d.). Redundant pointers, including extra successors and predecessors, serve as bulwarks against node failures, guaranteeing alternative paths for message routing and preserving the integrity of the ring structure.

Stabilization protocols stand as vigilant guardians, orchestrating periodic updates to successor pointers and safeguarding the network's structural coherence in the face of dynamic node behaviors (Stoica et al., n.d.). Leveraging the power of consistent hashing, Chord orchestrates a symphony of balanced distribution, evenly spreading IP addresses across the identifier space to foster load balancing and fault-tolerance (MIT Laboratory for Computer Science, n.d.). Furthermore, the ingenious implementation of finger pointers empowers Chord with optimized lookup operations, minimizing message hops and bolstering fault-tolerance through enhanced routing efficiency.

4.2.4 Using P2P technology in messaging and communication

P2P technology plays a crucial role in messaging and communication and offers several advantages over traditional centralized systems:

- Privacy and security: P2P messaging protocols ensure end-to-end encryption and direct communication between users, minimizing the risk of data breaches and unauthorized access by intermediaries (MIT Laboratory for Computer Science, n.d.).
- Direct interaction: P2P networks enable direct communication between users' devices without relying on centralized servers, allowing for real-time messaging and file sharing (MIT Laboratory for Computer Science, n.d.).
- Resilience: Decentralized P2P networks are inherently resistant to censorship and disruption, ensuring uninterrupted communication even in challenging environments (MIT Laboratory for Computer Science, n.d.).
- Efficiency: P2P messaging protocols optimize bandwidth usage and reduce latency by leveraging distributed resources, resulting in faster and more efficient communication (Stoica et al., n.d.).

4.2.4.1 Benefits and challenges

The use of P2P technology for messaging and communication offers several benefits, including improved privacy, security, resilience, and efficiency (Stoica et al., n.d.). However, challenges such as network complexity, security vulnerabilities and regulatory compliance need to be addressed to fully realize the potential in this area (Stoica et al., n.d.).

4.2.4.2 Real-world examples and case studies

Real-world examples illustrate the practical applications of P2P technology in messaging and communications:

- Signal: A P2P messaging app known for its strong focus on privacy and security using end-to-end encryption and direct communication between users' devices (MIT Laboratory for Computer Science, n.d.).
- Tox: An open source P2P messaging platform that prioritizes user privacy and freedom, offering communication without relying on centralized servers (Stoica et al., n.d.).
- Ricochet: A P2P instant messaging application that routes messages through a hidden Tor service, ensuring user anonymity and privacy (MIT Laboratory for Computer Science, n.d.).

4.3 TCP and UDP

TCP (Transmission Control Protocol) is a protocol centered around connections and security. The protocol works by executing a three-way handshake. First a listening server is requested for connection, thereafter that request is sent back to finally be acknowledged by the original

sender thus completing the connection. The connection is reliable and makes sure the information sent by the requester is received by the receiver thus making sure no data is lost (Kurose & Ross, 2018).

This stands in contrast to UDP (User Datagram Protocol) where the handshake is excluded from the process, and instead it is just set on the port and outputs constantly. Meaning that the packages get sent much faster and are easier to intercept which also leads to less overhead but also that it can be easily spoofed by malicious actors as the reliability and security of the handshake is not included (Kurose & Ross, 2018).

In conclusion a TCP connection is more secure, and the data sent is more reliably received while a UDP connection is much faster and easier to update with lower overhead while being less secure.

4.4 Data exchange formats

When transmitting data between different sources or just storing it, the format of the chosen object is important as that will define how easy it is to read, what is stored in the object, and how easy it is to transfer. All of this has to be considered as if the given application has an extraordinary amount of data needed to be transferred, then that part should be prioritized (Sriparasa, 2013). The easy and most simple way to do this is just having the data being unformatted. This of course means it is not standardized and harder to actually use, hence the need for a data format.

If unformatted data is not desired as in this case the system calls for a more readable format to humans, messages or the data needs to have a better format. Two of the most commonly used and most lightweight formats are JavaScript Object Notation, or JSON for short, and Extensible Markup Language also known as XML (Sriparasa, 2013).

The first, JSON, is a very lightweight notation that usually is converted to JavaScript objects when it is sent. It is formatted in key pairs, one is the key and the other is the value that key corresponds to. For these reasons, JSON is commonly used in web based architecture as it allows rapid scaling, while still being an efficient format for the data for both humans and machines (Sriparasa, 2013).

In contrast, XML is still a lightweight and easily readable format, but it is heavier and much more rigid than JSON (Sriparasa, 2013). This means it is more commonly used in more complex data formats like functions where more information needs to be passed on or where scaling is less important. XML as a data format is formatted as a tag and then the attributes related to that tag (Sriparasa, 2013).

4.5 Encryption

Encryption is a central part of cryptography and is an incredibly important aspect of data security. In a distributed network, the main part of security is using different security keys to

encrypt and decrypt data. This is important as if you cannot trust the encryption in a distributed system then the product becomes undependable, as if you cannot trust what information the system is giving you then you cannot use it effectively (Steen & Tanenbaum, 2017).

4.5.1 Symmetric and asymmetric

Symmetric and asymmetric are the two different ways of encryption that are used. They are also known as private and public encryption, as that is the main split between how the keys they use to encrypt are classified. The first kind of encryption is the private one, symmetric encryption (Steen & Tanenbaum, 2017).
Symmetric encryption works by having one private key shared by both parties. The sender of the encrypted data uses it to encrypt it and the receiver then uses the same key to decrypt it, thus making sure only the two people knowing the key can access it. This stands in contrast to the public way to do it - asymmetrical encryption (Steen & Tanenbaum, 2017). This way is called asymmetrical because unlike symmetrical, where both sides mirror each other by using the same key, asymmetrical encryption has two different keys, a public one that is used for encryption that is known to the public and a private key used for decrypting, only known to the decryptor. The main difference is that symmetrical encryption is faster and less hardware intensive to implement but has the problem of how to distribute keys as it is not inherent to the decryptors like in an asymmetrical decryption (Steen & Tanenbaum, 2017).

4.5.2 DES

DES is a symmetrical encryption algorithm that works on bits, where each group is segmented into base 16. As it is base 16 it means that the codes are written in hexadecimal so 0 1 E F responds to 0000 0001 1110 1111 (Grabbe, 2018). This means that when a message is sent it is encrypting groups of 64 message bits which is done by using 64 bits lengths keys that is in reality 56 bits as DES skips every 8th key bit.

As is explained every message needs to be 64 bits to fit into this neat system and gets sent properly and for this to work padding is needed. Padding is the practice of filling up the last bits until the string needing to be encrypted is 64 bits long, then padding is added to make sure that it fits this schema. This is done for extra security as it means the encrypted groups are harder to brute force and harder for humans to interpret and allow them to more easily guess the plaintext (Grabbe, 2018).

4.5.3 AES

AES, also known as Advanced Encryption Standard, is mostly just a mark improvement unto the DES algorithm. It was developed in response to the DES algorithm being publicly shown how easy it is to brute force. This is mainly solved with where DES is limited to a 64 bit block size and a key size of 56 bits the AES algorithm has block sizes of 128 bits and keys scaling all the way up to 256 bits. This means that it is far harder to brute force and for

outsiders to alter data using it. It manages to do this while being faster to both encrypt and decrypt with (Mahajan & Sachdeva, n.d.).

4.6 Hashing

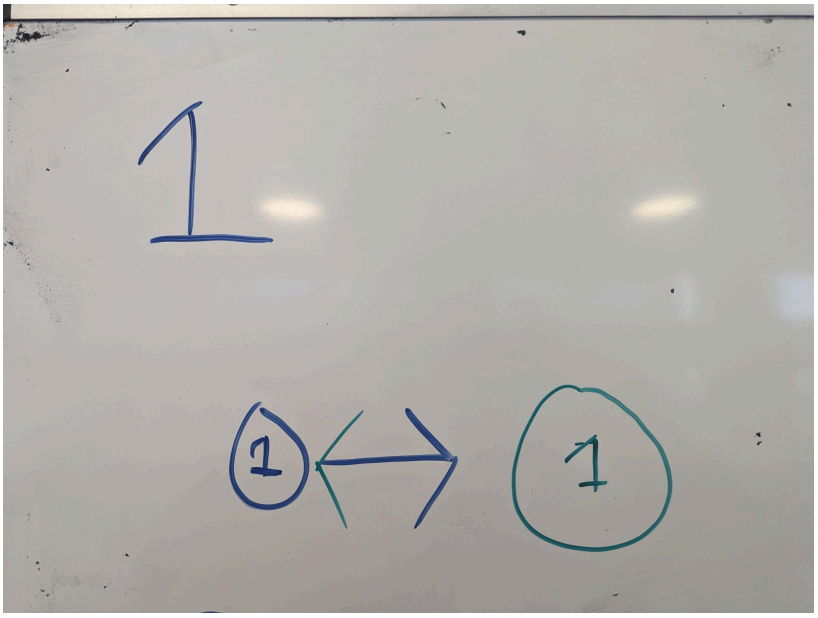
Hashing or as the specific use of hashing this project will use in encryption, “Unkeyed Cryptographic Hashing” is the practice of taking a string and then outputting it as an encrypted string of a predetermined size of bytes (Rivest, 2017). This is important as to secure data integrity by making sure that the original data still matches the stored data, while also protecting users' data in case of data leaks or hacking attempts. This encryption is usually easy to break as there is no key to it as implied by the name of the hashing category, thus making it incredibly easy to brute force (Rivest, 2017). Thus, many algorithms use padding and increase the number of bytes of the output. This while an improvement is still relatively easy to break through as machines are incredibly powerful. For these reasons it is recommended that the desired information is run through the hash encryption algorithm multiple times, for some up to the 10.000’s, to make sure that the information will be properly secure and that brute forcing is far harder (Rivest, 2017).

5 Design of our P2P-network

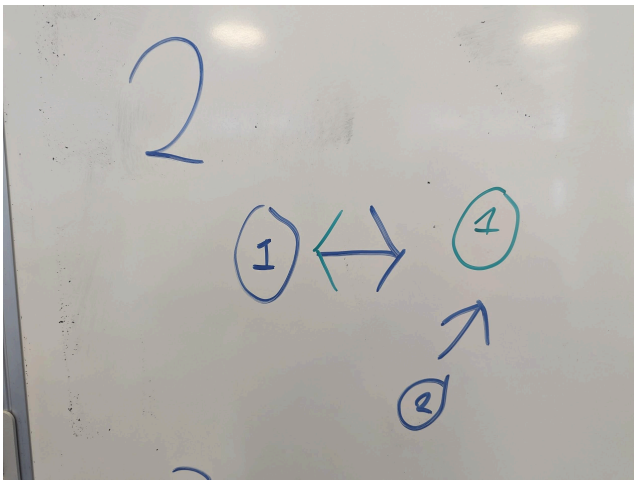
To document and properly explore our designs, we decided to track the different iterations and make sure that every thought and decision made while deciding on a design is documented. This will help explain how our application has evolved as a concept and why certain design decisions are taken.

5.1 First iteration

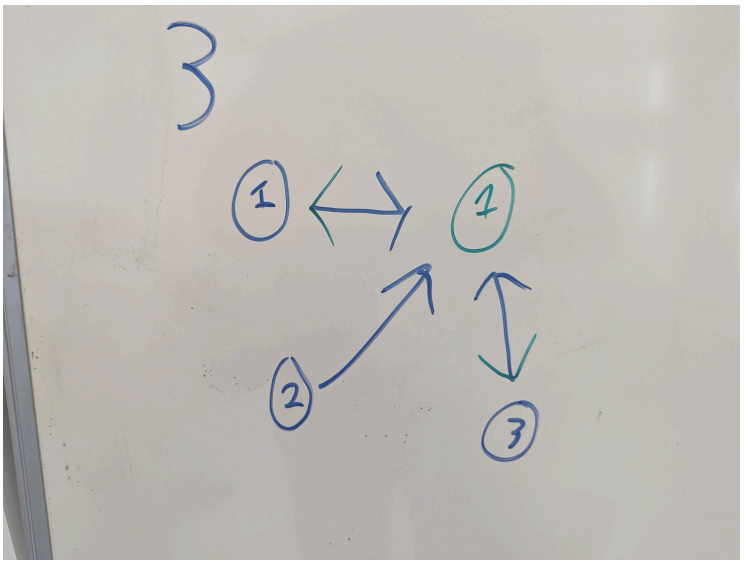
The design of the app consists of 5 different actions that the user must perform, which form the basis for all use of the app and all features that build on its functionality.
The first action that should be possible for the user is to create a P2P node. This node must be connected to the user who created it and hosted on their device.



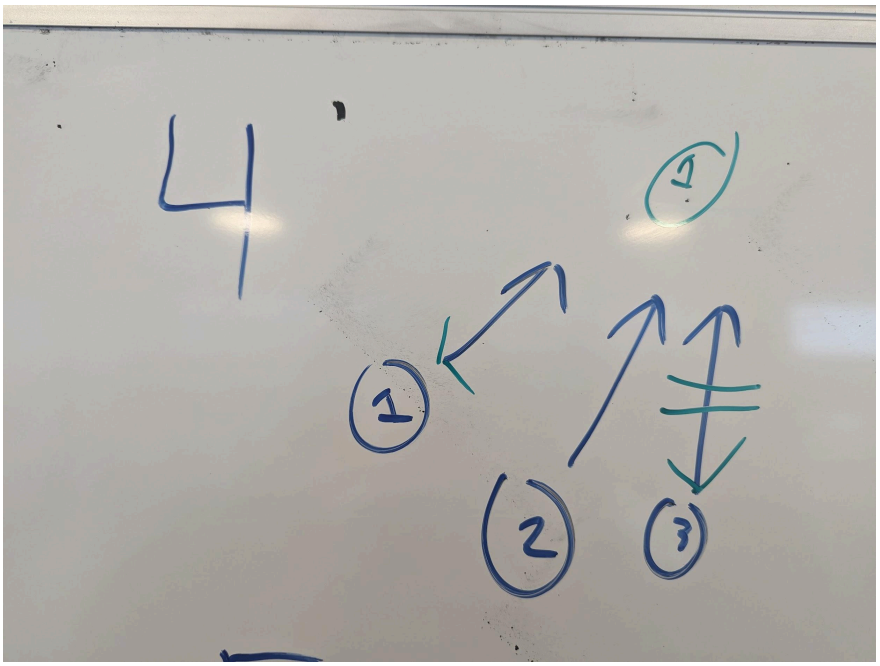
The second action that forms the basis is that other devices should be able to subscribe to the created node without creating a new node and thus access it. This subscription should include an option to simply view the content and an option to download some or all of it.



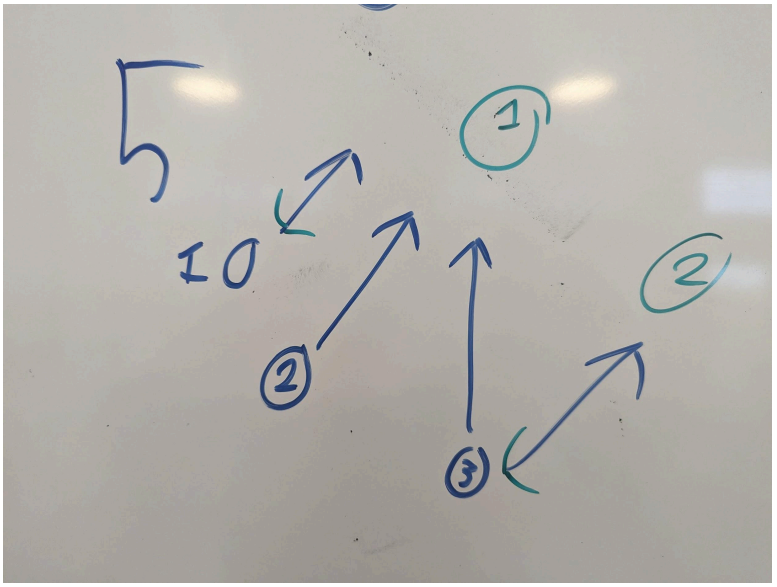
The third action is for a device other than the first to be able to mirror the node created by the first user. This is to make the nodes more robust, meaning that part of the P2P network is no longer completely dependent on one user.



The fourth action users should be able to perform is unlinking. This should not just be a feature for one mirror, but a universal action so that anyone can unlink. Meaning that if the original user wants to stop hosting the node, it can still be hosted by others and they can unlink it. Leading to some nodes will cease to function, but it can also be seen as a benefit as it will encourage more mirroring and active engagement in the P2P network.

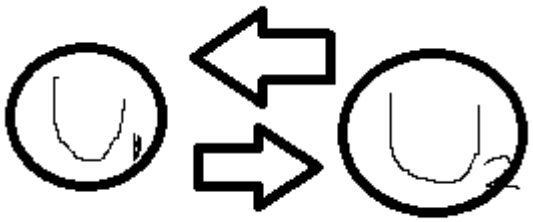


The final step is that a user should be able to create a new node even if they subscribe to a node that is already running, so it becomes a network and not just a bunch of isolated nodes.



To build this first draft, after some research, we decided to postpone the distributed database. Although it is an interesting concept, we decided that it could be a major source of delay and thus just create problems in achieving our main goal, which was to connect users with a P2P system. Therefore, we decided instead to start by implementing only a P2P communication network where users can connect to each other bilaterally.

5.2 Second iteration



To achieve the considerations made at the end of the first iteration, we decided that each user should have an ID to identify their individual node. Of course, this means the application runs into the cold start issue, where a user starts and has no way to get started using the application. To counter this, we had two general ideas: one is to show other IDs on the same network or users' names. The other is to show who other users have connected to in the network, so you can easily expand by knowing just one user.

For our core communication protocol, we mainly considered two different protocols, UDP (User Datagram Protocol) and TCP (Transmission Control Protocol), as they are the most common and universally used. This is used as a selection criterion as we will be able to easily integrate them into our design without much difficulty and therefore having already tested and widely used protocols makes the process easier. In our selection process we looked at

what we prioritized in our design and that was to ensure that we had a stable connection between the two nodes that the information is sent through, and that part speaks strongly for the use of TCP.

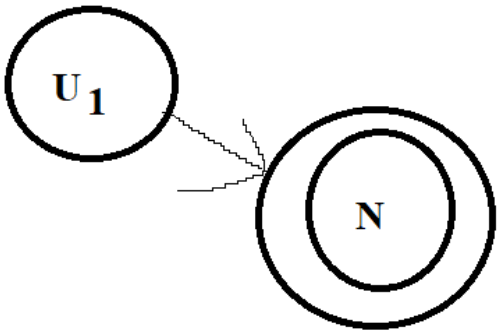
TCP has the clear advantage that it ensures that the message has been delivered and can therefore ensure that there are no problems with the connection. In the P2P messaging application that we currently prefer, a stable connection is needed, and therefore we should implement it with the current design. But having said that, UDP suddenly becomes more tempting in the future if we ever try to reconfigure it towards its original goal of being more of a database system, as there is now a reason why a node does not need to confirm if everything has been received, as errors that sometimes occur when forwarding data from servers wouldn't hurt anything major. With that in mind, it would also probably just be better to continue using TCP as there is no need for fast, constant transfer of information as it is not a game, but a messaging system.

That does not mean we do not use UDP; we use it to find the local nodes to connect to. We do this because they do not know what to connect to, and using UDP these connections can be made dynamically. Once the network is set up, the connections are secured and made reliable with TCP, so both are used, but the focus is still primarily on TCP for the actual exchange of messages.

5.3 Third iteration

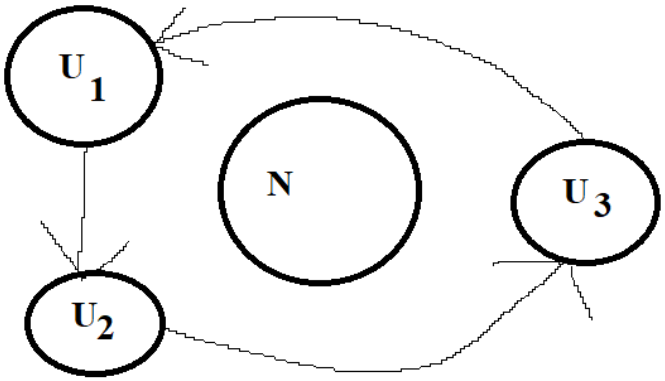
The second iteration does not provide a way to establish that UDP connection and therefore a third iteration of the design is needed. With the attempt to establish a TCP connection, we realized that this UDP connection is necessary, and therefore the next design focuses on allowing it.

The first step focused on is the same as in the first iteration, just slightly different. Instead of allowing a node in a cloud, the design should be able to allow the user to establish a node on a selected network.

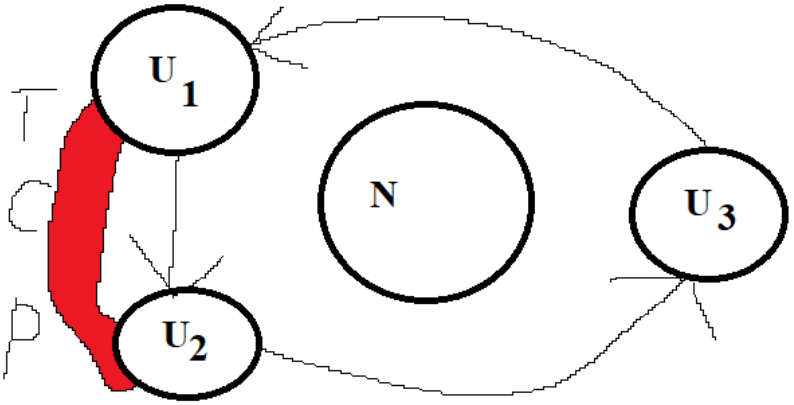


The second step is to allow other users to establish nodes, but instead of being forced to

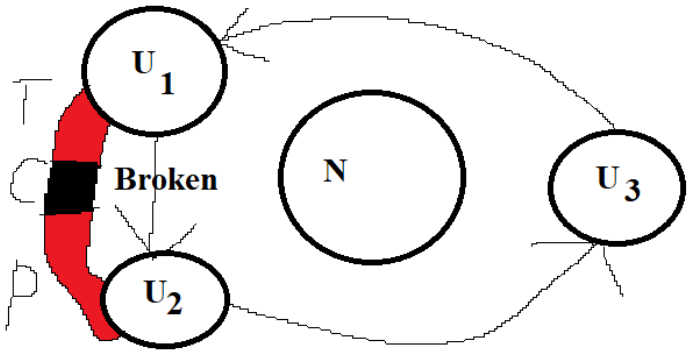
connect each node through a network as in the first iteration, the third iteration connects these nodes with a UDP connection. Once these connections are made, a Chord network of the nodes is established and a corresponding finger table is created to navigate it. Each time a new node is created in the same network, the node is added to the Chord network and the finger table is updated to include it.



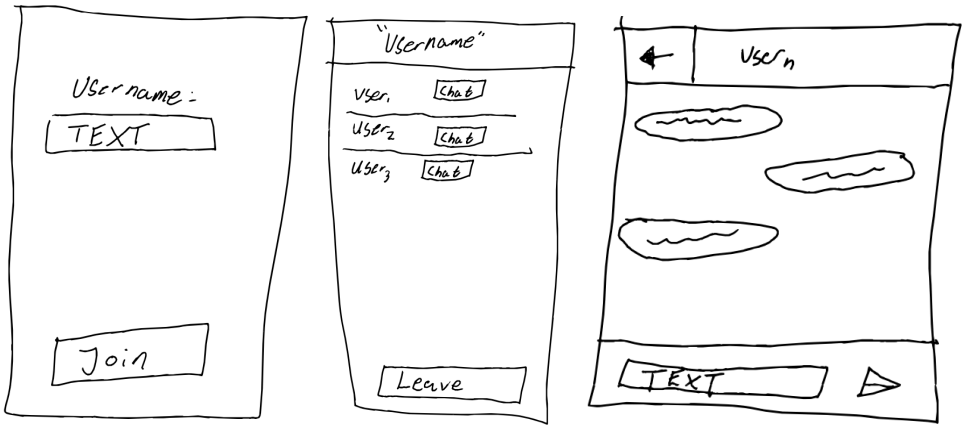
The third step is to allow two of the users on the Chord network to connect to each other. This connection is done with a TCP connection, ensuring that it is secure and that the information sent is reliable. To ensure that the information sent is secure and not just the connection, a symmetric encryption algorithm is developed between the two users, specifically AES. This is done because a symmetric encryption algorithm is preferred as the connection does not need to be public and AES is chosen as the encryption algorithm as it is more modern than DES while being fast to both encrypt and decrypt.



Finally, the last step in the design is to allow a user to disconnect their node from the network. Meaning that there is a way out if a user no longer wants to be part of the Chord network.



5.4 Wireframe



Our wireframe design philosophy emphasizes simplicity and intuitive usability. We aim to create an application that feels familiar to users of common chat apps, minimizing confusion and streamlining the user experience.

5.4.1 Step 1: Username Selection

The first step in our process is username selection. We provide a simple field where users can enter a username for their session. This design choice allows users the freedom to switch between different usernames across sessions, without being tied to a single identifier.

The only other feature on this screen is the "Join Network" button. This button is enabled only when a username has been entered, as a username is required to identify the user to others on the network.

5.4.2 Step 2: User Discovery

Once a user has joined the network, they are presented with a user discovery screen. This screen allows users to find other profiles, whether they are familiar contacts or new, interesting users.

To prevent user confusion, the current username is prominently displayed at the top of the screen. Below, a row of discoverable users is displayed, identified by their usernames. Users

can initiate a chat with a discovered user by clicking the "Chat" button next to their username.

A "Leave" button is also present, allowing users to end their current session and rejoin with a different username, or exit the application entirely.

5.4.3 Step 3: Chat Interface

The final screen in our wireframe is the chat interface. This screen is accessed once a user has initiated a chat and the other user has accepted.

A "Back" arrow is present in the top left, allowing users to return to the user discovery screen to find other chat partners. The username of the current chat partner is displayed to the right of the back arrow.

The chat history is displayed in the main body of the screen, allowing users to review their conversation. A text field at the bottom of the screen allows users to compose and send messages, with a send arrow to the right of the text field.

6 Analysis

This section provides an in-depth analysis of our application. It starts with an overview of the architecture, detailing how UDP is used for node discovery and CP for direct end-to-end communication. design principles and patterns such as modularity and reuse are highlighted. Core features and user interface design are examined in detail. The section concludes with a review of unit testing results, unresolved issues and an assessment of potential security threats.

6.1 Introduction (a brief overview)

In this project, we set out to design and implement a communication network that operates as a peer-to-peer (P2P) network without any central server intervention. This approach is particularly intriguing because it allows us to create a network that is entirely controlled by its users, free from the influence or oversight of the creators. The objective is to leverage elements of the Chord protocol to facilitate node organization and communication within a decentralized network.

The importance of P2P networks lies in their ability to distribute resources and workload among all participants, reducing the risk of a single point of failure and enhancing the system's robustness and scalability. The Chord protocol is crucial in distributed systems as it provides an efficient method for maintaining node connections and network structure, which is fundamental for the performance of a P2P network.

Given the complexity of designing and implementing such a network from scratch in only a few months, we established several boundaries to keep the project manageable. First, we decided the network would be limited to a local network, meaning all devices (nodes) need to be connected to the same network. This allowed us to focus our resources on making the communication between nodes work effectively without dealing with broader network issues.

Secondly, we chose to restrict the communication within the network to text-based messages. This decision enabled us to concentrate on the communication protocol itself rather than dealing with the complexities of multimedia data transformation.

Third, we aimed to create an actual application with an interactive user interface, moving beyond just a console application. We chose to develop a mobile application for Android using Android Studio, primarily because of our prior experience with the platform. However, this choice was arbitrary, and we could have just as easily opted for a desktop application.

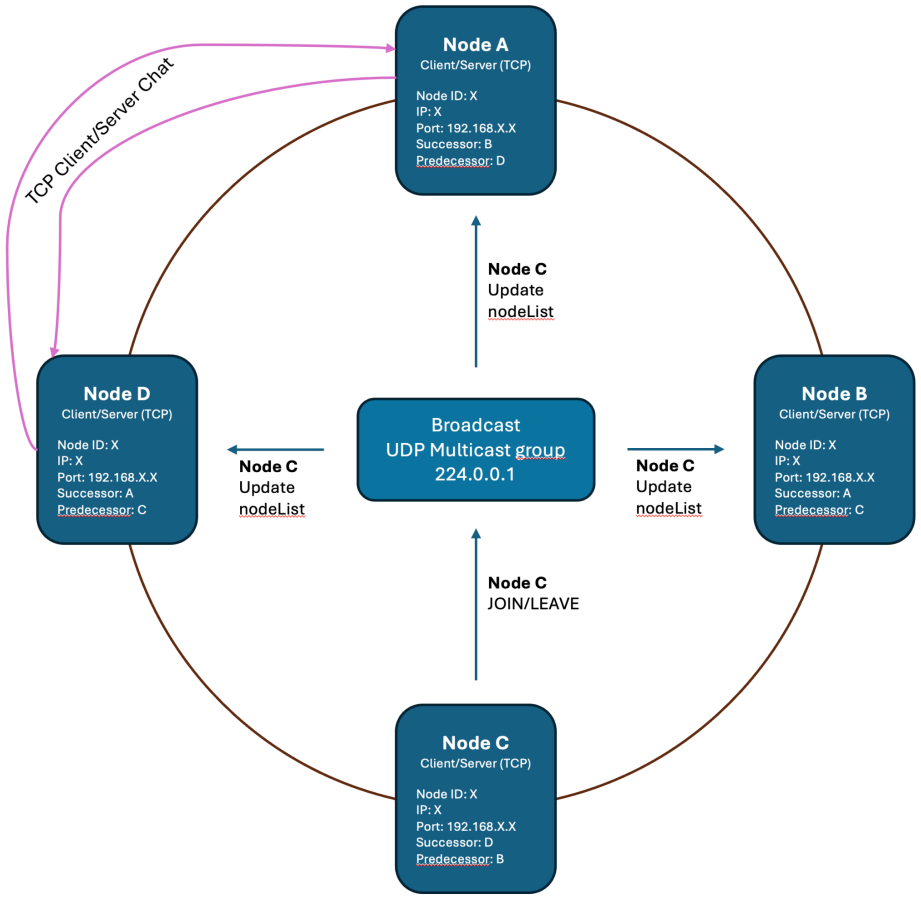
Lastly, we committed to a 100% serverless architecture. This decision presents both a boundary and a challenge, requiring us to devise methods for node discovery and communication without a centralized lookup mechanism. We implemented a discovery broadcast using UDP, allowing nodes to find each other on the local network. Once discovered, nodes utilize the Chord ring to identify their predecessor and successor. Each

node has a unique ID, hashed from its IP address and port number. Nodes then establish direct TCP connections for end-to-end chat communication.

Throughout the implementation, we encountered numerous challenges, including cold start problems, network issues, and the proper setup of the Chord structure to add value to our implementation. In the following sections, we will delve into various mechanisms and concepts of the code, the challenges that arose from these components, and how we addressed them.

6.2 Architecture

In this section, we provide an overview of the application's architecture, focusing on how different components interact and communicate. This includes the networking mechanisms used for node discovery and communication, the implementation of the Chord protocol for managing the distributed network, and the data management strategies implemented for the chat functionality. This holistic view highlights the interconnected nature of these components and their roles in ensuring the application's functionality.



6.2.1 Networking and Chord implementation

The networking component of the application is crucial for establishing communication between nodes in the distributed network. Nodes announce their presence using UDP

multicast, allowing them to discover each other efficiently. Once discovered, nodes establish direct connections using TCP for more reliable and persistent communication.

The *MulticastService* class handles the UDP multicast communication. It sends and receives multicast messages to announce the presence of nodes. When a node joins the network, it sends a *JOIN* message containing its details. Similarly, when a node leaves, it sends a *LEAVE* message. The service also listens for incoming multicast messages to update the list of active nodes accordingly. This initial discovery phase is critical for forming the initial network structure and ensuring that all nodes are aware of each other.

Once nodes are aware of each other through multicast messages, they have the opportunity to establish direct TCP connections for reliable communication. The *ChatService* class manages these TCP connections. It initiates connection to other nodes, listens for incoming messages and ensures that data is transmitted reliably. The *ChatServerService* runs a server socket to accept incoming TCP connections from other nodes. This dual approach, using UDP for discovery and TCP for reliable communication, ensures both flexibility and robustness of the network.

Our application incorporates elements of the Chord protocol. Each node in the network has a unique identifier (node ID) derived from its IP address and port using SHA-1 hashing. This node ID is used to position the node within the Chord ring, determining the successor and predecessor, facilitating the organization of the network.

The *StabilizationService* ensures the network's consistency and robustness by periodically checking and updating the successor and predecessor pointerns of nodes. It sends multicast messages to inform other nodes of its presence. This dynamic adjustment mechanism is key to maintaining the integrity and performance of the network over time.

6.2.2 Data management and chat functionality

The data management component is primarily focused on the chat functionality, allowing nodes to send and receive messages within the network. This is facilitated through the *Message* class, which represents messages with their content, timestamp and the sender's IP address. This standardized message format ensures consistency and ease of processing across the network.

The *ChatService* class handles the sending and receiving of these messages over TCP connections. When a message is received, it is passed to the registered *MessageListener* to update the user interface accordingly. This real-time communication capability is critical for maintaining the interactivity and responsiveness of the chat feature.

The *ChatActivity* class provides the chat interface where users can send and receive messages. It interacts with the *ChatService* to send messages typed by the user and display incoming messages in real time. This seamless integration between the user interface and the

underlying network services ensures a smooth user experience, allowing users to focus on their interactions rather than the technical details of message transmission.

The *ChatServerService* manages the server-side aspects of the chat functionality, including accepting incoming connections and forwarding received messages to the appropriate *ChatActivity* instances. This separation of concerts ensures that the application potentially can handle multiple chat sessions simultaneously, maintaining a smooth user experience even under varying network conditions.

Overall, the application leverages a combination of UDP for initial discovery and TCP for reliable communication, incorporating elements from the Chord protocol. This comprehensive approach ensures robust and scalable communication between nodes, enabling efficient data management and real-time chat functionality. The integration of these components provides a solid foundation for the application's performance and usability, ensuring that it meets the needs of requirements while maintaining flexibility and reliability in a dynamic network environment.

6.3 Design principles and patterns

In software development, adhering to well-established design principles and patterns is essential for creating robust, scalable and maintainable applications. These practices have provided us with a more structured approach to the development of our application, ensuring that the code is organized, reusable and easier to understand. By adhering to the principles and patterns mentioned below, we have created a system that is modular, allowing for individual components to be developed, tested and maintained independently. This modularity not only facilitates easier debugging and testing, but also simplifies the process of adding new features or modifying existing ones.

6.3.1 Separation of Concerns

The design principle Separation of Concerns (SoC) divides a software system into distinct sections, each handling a specific aspect of functionality. This reduces complexity and enhances maintainability by allowing individual parts to be developed, tested and debugged independently.
This principle has been applied throughout our application to enhance modularity and maintainability. For instance, the network communication logic is separated into the *MulticastService* and *ChatService* classes, while the user interface logic is handled by various activity classes such as *MainActivity* and *ChatActivity*.

6.3.2 Single Responsibility Principle

The Single Responsibility Principle (SRP) states that a class should have only one reason to change, meaning it should have only one job or responsibility. This principle simplifies the code and makes it easier to understand, test and maintain.
The *ChatActivity* class in our application is responsible solely for managing the chat interface and interactions, ensuring the chat logic is isolated from other functionality.

6.3.3 Encapsulation

Encapsulation hides the internal state and behavior of an object and only exposes a controlled interface. This protects the integrity of the data and reduces interdependencies between components.
The *NodeInfo* class encapsulates nodes' properties, providing getters and setters to access and modify node information safely.

6.3.4 Singleton

The Singleton pattern is used to ensure that a class has only one instance and provides a global point of access to it. In our application, the *MulticastService* and *ChatService* classes are implemented as singletons. This approach ensures that there is a single instance managing the network communications and chat functionalities, respectively, preventing conflicts and ensuring consistent behavior across the application.

6.3.5 Adapter

The adapter pattern allows incompatible interfaces to work together by converting the interfaces of a class into another interface expected by the clients. This facilitates the integration of new components without altering existing code.
The *NodesAdapter* class adapts the list of nodes to be displayed in a RecyclerView UI component, making it compatible with the Android UI framework. On line 15 in the code-snippet below, the *NodesAdapter* class extends *RecycleView.Adapter[...]* and is responsible for managing the data and creating/binding view holders for the *RecyclerView* to display a list of *NodeInfo* objects.

```
4 usages  jatindershub
15  public class NodesAdapter extends RecyclerView.Adapter<NodesAdapter.NodeViewHolder> {
        3 usages
16      private List<NodeInfo> nodes;
        2 usages
17      private OnItemClickListener listener;
```

6.3.6 Service

The service pattern defines a class that provides a specific set of functionalities. It is used to encapsulate business logic and can be invoked independently of the client that requests it. This is particularly useful for background operations and ensures a clear separation between the user interface and business logic.
In our application there are three classes that follow this design pattern; *ChatService*, *MulticastService*, and *StabilizationService*. *ChatService* provides background services for chat functionality, allowing the application to handle chat operations asynchronously. *MulticastService* manages multicast communication, enabling efficient message distribution across multiple nodes. *StabilizationService* periodically stabilizes the Chord ring by verifying and correcting successor and predecessor pointers, maintaining the integrity of the distributed system.

In our project, the synergy of these principles and patterns has provided a solid foundation, ensuring that each component can operate independently while still contributing to the overall functionality of the system. This approach has reduced complexity and improved code readability, making the development process more efficient. Through this structured methodology, our application benefits from a design that promotes both current stability and future flexibility.

6.4 Core features

In this section, we will delve into the core features of our application. For each feature, we will begin by presenting a sequence diagram to illustrate the interactions and message flow involved in the component's functionality. Following the sequence diagram, we will provide a detailed explanation of the corresponding code, highlighting how the implementation achieves the described behavior. This structured approach will offer a comprehensive understanding of each core component, demonstrating both the design and practical implementation within our application.

6.4.1 Prerequisites

To fully understand the following analysis, it is essential to have a basic familiarity with Android development concepts, particularly the lifecycle methods such as *onCreate* and *onDestroy*. These methods are fundamental to managing the behavior and state of Android activities and services.

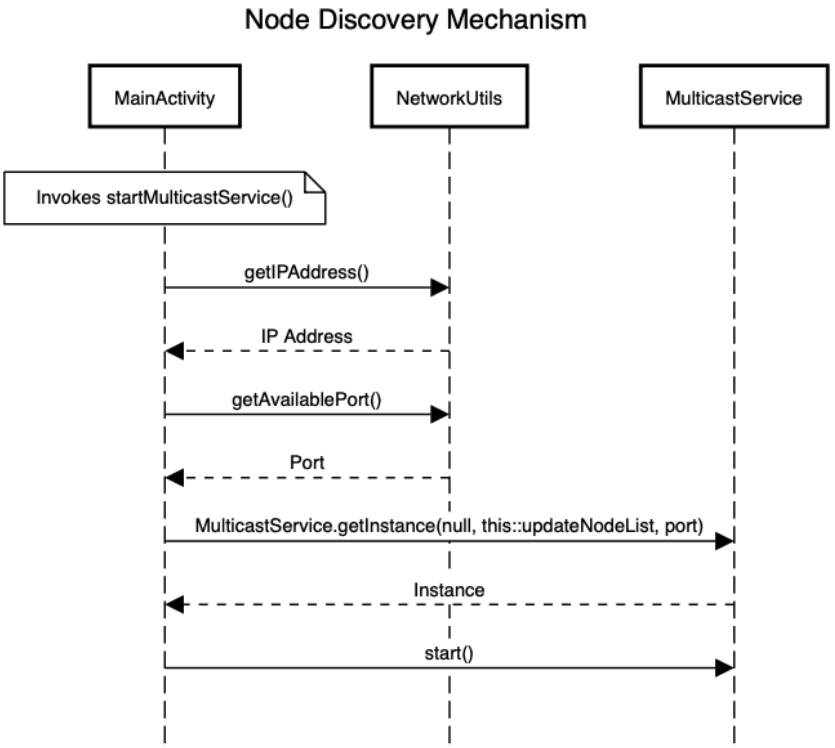
***onCreate* method**
The *onCreate* method is a crucial part of the Android activity lifecycle, as it is called when the activity is first created. This method is where all the one-time initialization for activities gets performed, such as setting up the user interface, initializing variables, and configuring components like buttons and RecyclerViews.

***onDestroy* method**
The *onDestroy* method is another important part of the Android activity lifecycle, as it is called before the activity is destroyed. This is the final call that the activity receives, allowing it to clean up any resources it holds before the activity is actually destroyed.

If the application crashes suddenly, the *onDestroy* method is not guaranteed to be called. This is because a crash is an unexpected termination of the application process, and the system may not have the opportunity to invoke the normal lifecycle callbacks.

6.4.2 Node discovery mechanism

The following sequence diagram shows the methods that together manage the initialization and execution of the multicast service, facilitating node discovery and communication in the application.



***MainActivity* class**

```
1 usage   jatindershub *
162     private void startMulticastService() {
163         InetAddress ip = NetworkUtils.getIPAddress();
164         int port = NetworkUtils.getAvailablePort();
165
166         // Conditional statement checks if the IP and port are valid
167         if (ip != null && port != -1) {
168             multicastService = MulticastService.getInstance( localNode: null, this::updateNodeList, port);
169             multicastService.start();
170         } else {
171             runOnUiThread(() -> nodeStatus.setText("Failed to get IP address or port"));
172         }
173     }
174
```

The method *startMulticastService*, which is inside the *onCreate* method, is responsible for starting the multicast service. The method first retrieves the local IP address and available port (on line 163 and 164) using the methods *getIPAddress* and *getAvailablePort* from the *NetworkUtils* class. The method then checks (on line 167) if both the IP address and port are valid. If the IP address and port are valid, the method retrieves an instance of *MulticastService* using the method *getInstance*. It then starts the multicast service on line 169.

If either the IP address or port is invalid, the method updates the UI on the main thread, on line 171, to inform the user that it failed to get the IP address or port. The methods which support the *startMulticastService* method are mentioned below.

***NetworkUtils* class**

```
2 usages  jatindershub
public static InetAddress getIPAddress() {
    try {
        for (NetworkInterface ni : Collections.list(NetworkInterface.getNetworkInterfaces())) {
            for (InetAddress address : Collections.list(ni.getInetAddresses())) {
                if (!address.isLoopbackAddress() && address.isSiteLocalAddress()) {
                    return address;
                }
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

2 usages  jatindershub
public static int getAvailablePort() {
    try (ServerSocket socket = new ServerSocket( port: 0)) {
        return socket.getLocalPort();
    } catch (IOException e) {
        e.printStackTrace();
        return -1; // Indicate an error if no port is found
    }
}
}
```

The *getIPAddress* method is a static method designed to retrieve the local IP address of the device. On line 13 the method *getNetworkInterface* provides an enumeration of all available network interfaces on the device, which is converted to a list for easier iteration. For each network interface the method *getInetAddresses* returns an enumeration of all IP addresses associated with that interface, which again converts this enumeration into a list for iteration (line 14). The method checks each IP address to ensure it is not a loopback address, for instance 127.0.0.1, which is used for internal communication within the device, or if the address is a site-local address (typically used in private networks, for instance 10.x.x.x). If an IP address meets both conditions, it is returned as a valid IP address (line 16). However, if an exception occurs during the process, it is caught on line 20, and the stack trace is printed. Therefore, if no valid IP address is found, the method returns null (on line 23).

The *getAvailablePort* method is also a static method designed to find and return an available port number on the local device. On line 27 the method attempts to create an instance of *ServerSocket* with the port number set to “0”. This instructs the operating system to allocate an available port. Once the *ServerSocket* is successfully created, the method retrieves the port number using the method *getLocalPort* and returns it (on line 28). The try-with-resources statement ensures that the *ServerSocket* is automatically closed when the block exists, whether normally or due to an exception. This is important to free up the allocated port and avoid resource leaks.
If an *IOException* occurs during the creation of the *ServerSocket*, the method catches the exception (on line 29) and prints the stack trace for debugging purposes, and returns the value “-1” to indicate an error.

***MulticastService* class**

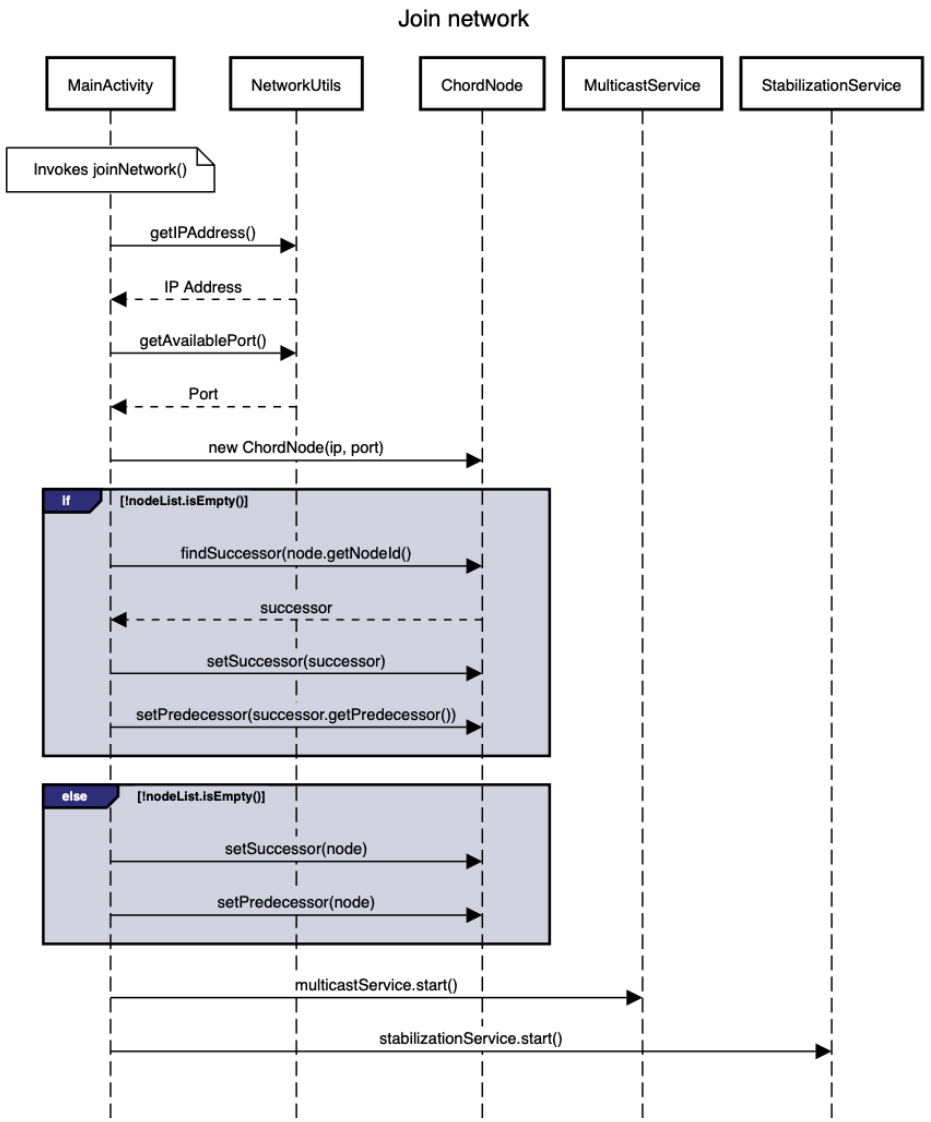
```
2 usages
37      private static final int PORT = 5000;
        1 usage
38      private static final String MULTICAST_ADDRESS = "224.0.0.1";
        2 usages  jatindershub
39      public MulticastService(ChordNode localNode, Consumer<List<NodeInfo>> nodeListUpdater, int port) {
40          this.localNode = localNode;
41          this.nodeListUpdater = nodeListUpdater;
42          this.dynamicPort = port;
43          nodeList = new ArrayList<>();
44          try {
45              socket = new MulticastSocket(PORT);
46              group = InetAddress.getByName(MULTICAST_ADDRESS);
47              socket.joinGroup(group);
48          } catch (Exception e) {
49              e.printStackTrace();
50          }
51      }
52
        1 usage  jatindershub
53      public static synchronized MulticastService getInstance(ChordNode localNode, Consumer<List<NodeInfo>> nodeListUpdater, int port) {
54          if (instance == null) {
55              instance = new MulticastService(localNode, nodeListUpdater, port);
56          }
57          return instance;
58      }
```

The *MulticastService* constructor on line 39 sets up the service by first initializing its parameters *localNode, nodeListUpdater* and *dynamicPort*. On line 34 an empty list gets initialized to keep track of discovered nodes. Afterward creating a multicast socket (on line 45), resolving the multicast group address (on line 46), joining a multicast group (on line 47) and preparing to receive multicast messages. This setup allows the service to discover other nodes on the network and maintain the node list.

The *getInstance* method on line 53 ensures that only one instance of the *MulticastService* is created by using the singleton pattern. The method is synchronized (as seen on line 53) to make it thread-safe, ensuring that only one thread can execute this method at a time. If an *instance* is null, it creates a new instance of *MulticastService*. The method returns the singleton instance of *MulticastService* on line 57.

6.4.3 Join network

This feature allows the node to join the network efficiently, update the UI appropriately and start the necessary services for network communication and chat functionality. The method also maintains the Chord ring structure.



***MainActivity* class**

```
4 usages
50      private Button joinNetworkButton;
51
        jatindershub *
52      @Override
53      protected void onCreate(Bundle savedInstanceState) {
54          super.onCreate(savedInstanceState);
55          setContentView(R.layout.activity_main);
56
57          nodeStatus = findViewById(R.id.nodeStatus);
58          joinNetworkButton = findViewById(R.id.joinNetworkButton);
```

The *joinNetworkButton* variable gets declared of type Button on line 50. The variable then finds the button view in the UI using its ID *joinNetworkButton* through the *findViewById* class on line 58 and assigns it to the declared variable.

```
115
116         joinNetworkButton.setOnClickListener(v -> new Thread(this::joinNetwork).start());
117         leaveNetworkButton.setOnClickListener(v -> new Thread(this::leaveNetwork).start());
118         viewDetailsButton.setOnClickListener(v -> viewNodeDetails());
```

On line 116 there is an *OnClickListener set* on the joinNetworkButton, which means that when the button is clicked, it starts a new thread to execute the *joinNetwork* method.

```
        1 usage  jatindershub
218     private void joinNetwork() {
219         new Thread(() -> {
220             try {
221                 InetAddress ip = NetworkUtils.getIPAddress();
222                 int port = NetworkUtils.getAvailablePort();
223
224                 if (ip != null && port != -1) {
225                     node = new ChordNode(ip, port);
```

```
259                 } else {
260                     runOnUiThread(() -> nodeStatus.setText("Failed to get IP address or port"));
261                 }
```

The *joinNetwork* method handles the logic for a node joining the network. It is executed in a separate thread to avoid blocking the UI thread. This can be seen on line 219.
On line 221 and 222 the methods *getIPAddress* and *getAvailablePort* are called from the *NetworkUtils* class, which retrieve the IP address and an available port. These methods are described in depth in the Node discovery mechanism. On line 224 the IP address and port are validated, and if they are not valid, the *nodeStatus* gets set to “Failed [...]” on line 260.

```
227             // Try to find an existing node to join the network
228             if (!nodeList.isEmpty()) {
229                 NodeInfo bootstrapNodeInfo = nodeList.get(0); // Assuming the first node in the list as bootstrap
230                 ChordNode bootstrapNode = new ChordNode(bootstrapNodeInfo);
231
232                 ChordNode successor = bootstrapNode.findSuccessor(node.getNodeId());
233                 node.setSuccessor(successor);
234                 node.setPredecessor(successor.getPredecessor());
235                 successor.setPredecessor(node);
236             } else {
237                 // This is the first node in the network
238                 node.setSuccessor(node);
239                 node.setPredecessor(node);
240             }
```

If the *nodeList* is not empty, we assume the first node in the list (index 0) is an anchor node. We then create the object *anchorNode* of type *ChordNode* and assign the values from the first node to this node (line 229 and 230). Afterward we create the object *successor* of type *ChordNode* and assign the values from the *anchorNode*'s successor (line 232 and 233). The node that initiated the *joinNetwork* component gets set as the predecessor on line 235.

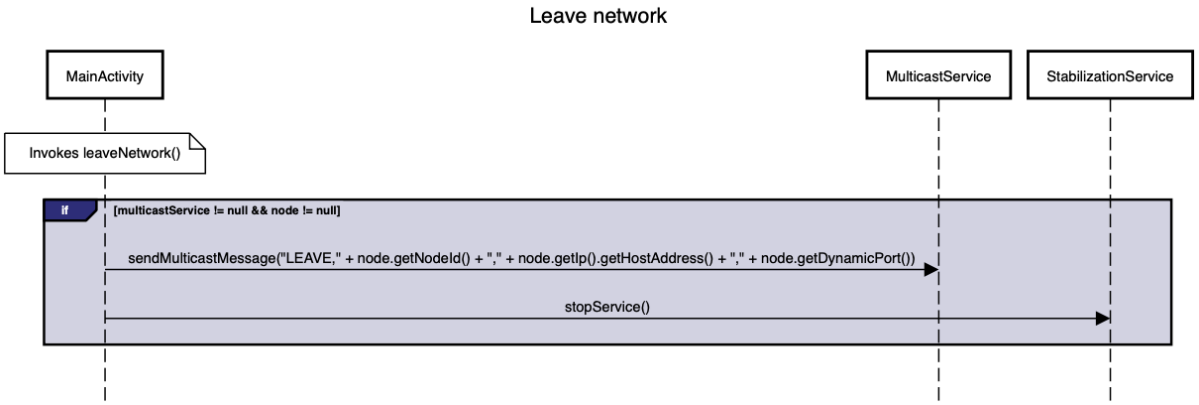
If the *nodeList* is empty, we assume that this is the first node in the network, and sets the node as successor and predecessor (line 238 and 239).

```
242             // Update multicastService to use the new node
243             multicastService = new MulticastService(node, this::updateNodeList, port);
244             multicastService.start();
245
246             stabilizationService = new StabilizationService(node, multicastService);
247             stabilizationService.start();
248
249             runOnUiThread(() -> {
250                 nodeStatus.setText("Node ID: " + node.getNodeId());
251                 viewDetailsButton.setEnabled(true);
252                 leaveNetworkButton.setEnabled(true);
253                 joinNetworkButton.setEnabled(false);
254             });
```

```
257
258             multicastService.sendMulticastMessage("JOIN," + node.getNodeId() + "," + ip.getHostAddress() + "," + port);
```

On line 243 and 244 we update the *MulticastService* to use the new node. The *MulticastService* gets initialized with the given local node, a reference to the method *updateNodeList* that updates the *nodeList* and finally the newly retrieved available port number. The *MulticastService* is responsible for sending and receiving multicast messages to discover other nodes on the network.
On line 246 and 247 the *StabilizationService* gets initialized with the given local node and the multicast service. The service is responsible for periodically checking and updating the local node's successor and predecessor references to ensure the network remains stable and consistent. This ensures the Chord ring structure is maintained even as nodes join or leave the network.
Both the *MulticastService* and *StabilicationService* get started with the *start* method in a separate thread, since the *joinNetwork* method extends *Thread*.
On line 249 to 254, the UI gets updated to reflect the node's status on the main thread. We delve more into that in the section UI components.
Finally, the new node's presence is announced through a multicast message to notify other nodes in the network (line 258).

6.4.4 Leave network



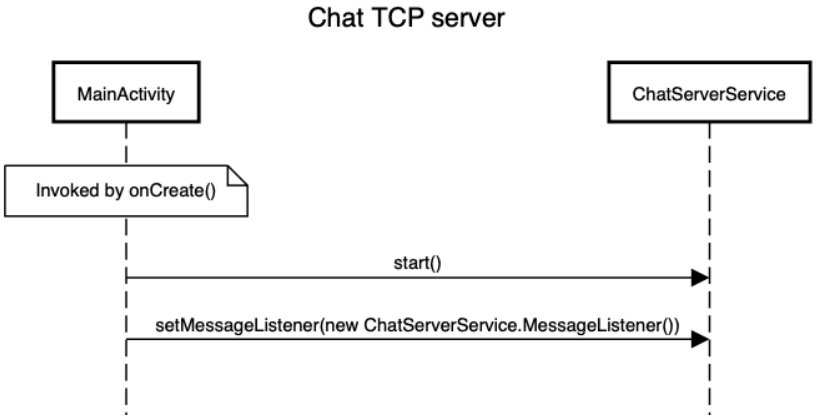
The *leaveNetwork* method effectively manages the process of a node leaving the network by sending a notification, stopping necessary services, and updating the UI to reflect the changes. The primary actors involved are the user (initiating the leave network action, explained in a later section) and the *MainActivity* class.

***MainActivity* class**

```
1 usage   jatindershub
private void leaveNetwork() {
    if (multicastService != null && node != null) {
        multicastService.sendMulticastMessage("LEAVE," + node.getNodeId() + "," + node.getIp().getHostAddress() + "," + node.getDynamicPort());
        stabilizationService.stopService();
        runOnUiThread(() -> {
            nodeStatus.setText("Node has left the network.");
            viewDetailsButton.setEnabled(false);
            leaveNetworkButton.setEnabled(false);
            joinNetworkButton.setEnabled(true);
        });
    }
}
```

The method starts by checking if both *multiastService* and *node* are not null to ensure they are properly initialized. If the checks pass, it sends a multicast message to notify other nodes that the current node is leaving the network. The message includes the node ID, IP address and dynamic port. The method then stops the *stabilizationService*, which is responsible for maintaining the network's stability and consistency. Finally, the UI is updated to reflect that the node has left the network. This is done on the main thread using *runOnUiThread* to ensure UI updates are made safely.

6.4.5 Chat TCP server



The startServer method in MainActivity initializes the server setup by creating a new thread that calls the method startServer in the ChatServerServices. The *ChatServerService* then initializes a *ServerSocket* to listen for incoming client connections on a specified port. When a client connects, it accepts the connection and spawns a new thread to handle the client's communication, allowing the server to manage multiple clients concurrently.

***MainActivity* class**

```
48      @Override
49      protected void onCreate(Bundle savedInstanceState) {
50          super.onCreate(savedInstanceState);
51          setContentView(R.layout.activity_main);
```

```
64          // Start the TCP server
65          startServer();
```

The method *startServer* is invoked on line 65 inside of the *onCreate* method in *MainActivity*, and initiates the process of starting a server.

```
        1 usage   jatindershub
109     private void startServer() {
110         chatServerService = new ChatServerService( context: MainActivity.this, SERVER_PORT);
111         chatServerService.start();
            jatindershub
112         chatServerService.setMessageListener(new ChatServerService.MessageListener() {
                1 usage   jatindershub
113             @Override
114             public void onMessageReceived(Message message) {
115                 // Handle received message
116                 Log.d( tag: "MainActivity",  msg: "Received message: " + message.getMessage());
117             }
118         });
119     }
```

The method is implemented in *MainActivity* on line 109 to 119. The method starts by creating an instance of *ChatServerService* on line 110 and thereby creates an *Intent* to start

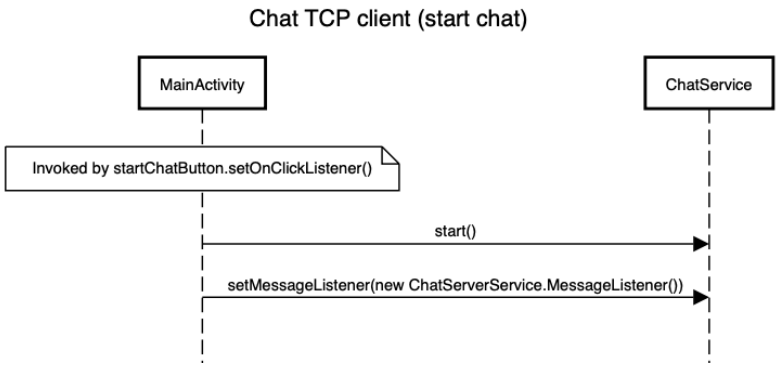
the *ChatServerService* with a specific port defined inside of the *SERVER_PORT* property. The method's primary purpose is to initiate the *ChatServerServices* to listen for incoming chat connections on the port mentioned before.

***ChatServerService* class**

```
1 usage  jatindershub
public void start() {
    new Thread(() -> {
        try {
            serverSocket = new ServerSocket(port, backlog: 0, InetAddress.getByName( host: "0.0.0.0"));
            Log.d( tag: "ChatServerService",  msg: "Server started on port " + port);

            while (true) {
                clientSocket = serverSocket.accept();
                String clientAddress = clientSocket.getInetAddress().getHostAddress();
                Log.d( tag: "ChatServerService",  msg: "Client connected from " + clientAddress);

                input = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
                output = new PrintWriter(clientSocket.getOutputStream(),  autoFlush: true);

                // Get the server's IP address
                String serverAddress = serverSocket.getInetAddress().getHostAddress();

                // Notify the MainActivity that the connection is established and start ChatActivity
                mainHandler.post(() -> {
                    Intent intent = new Intent(context, ChatActivity.class);
                    intent.putExtra( name: "ipAddress", serverAddress);
                    intent.putExtra( name: "port", port);
                    context.startActivity(intent);
                });

                listenForMessages();
            }
        } catch (IOException e) {
            Log.e( tag: "ChatServerService",  msg: "Error starting server", e);
        }
    }).start();
}
```

The *start* method starts by creating a new thread using a lambda expression. This is to run the server socket operations in a separate thread, ensuring that the main thread of the application remains responsive. On line 40 a *serverSocket* object is created to listen for incoming connections. The socket is bound to the IP address "0.0.0.0", which means it listens on all available network interfaces. On line 43 the server enters a n infinite loop to continuously accept incoming client connections. When a client connects, the server creates a *Socket* object for the client. For each connected client, the server sets up input and output streams to read from and write to the client (line 48 and 49). The server uses a handler *mainHandler* to post a runnable to the main thread on line 55. This runnable creates an intent to start the *ChatActivity* class by passing the server's IP address and port as extras. After setting up the connection and notifying the *MainActivity*, the server calls the method *listenForMessages* on line 62. If an *IOException* occurs at any point during the server's operation, it is caught and logged.

6.4.6 Chat TCP client (start chat)



MainActivity sets up a button click listener to extract the IP and port, create and start a *ChatService* instance, and log incoming messages via a listener. *ChatService* connects to the server, sets up communication streams, starts *ChatActivity*, and listens for messages in a separate thread, forwarding them to the listener.

***MainActivity* class**

```
68          startChatButton.setOnClickListener(new View.OnClickListener() {
                jatindershub
69              @Override
70              public void onClick(View view) {
71                  String ipAddress = ipAddressInput.getText().toString();
72                  int port = Integer.parseInt(portInput.getText().toString());
73
74                  // Connect to the server (self or other device)
75                  chatService = new ChatService( context: MainActivity.this, ipAddress, port);
76                  chatService.start();
                    jatindershub
77                  chatService.setMessageListener(new ChatService.MessageListener() {
                        1 usage  jatindershub
78                      @Override
79                      public void onMessageReceived(Message message) {
80                          // todo: handle receuved message
81                          Log.d( tag: "MainActivity",  msg: "Received message: " + message.getMessage());
82                      }
83                  });
84              }
85          });
```

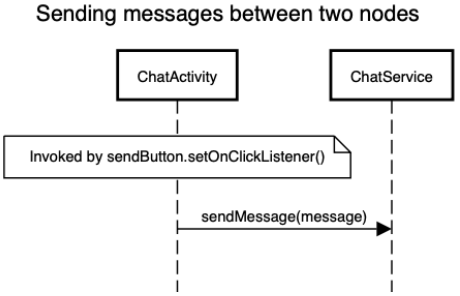
The *startChatButton* (of type *Button*) is configured with an *OnClickListener* that defines what happens when the button is clicked (line 68). The IP address and port number are extracted from the user input fields *ipAddressInput* and *portInput* on line 71 and 72. A new instance of *ChatService* is created on line 75 with the context, IP address and port. The *start* method of *ChatService* is called to initiate the connection on line 76. Finally, there is a message listener to handle incoming messages and log them.

***ChatService* class**

```
1 usage  jatindershub
public synchronized void start() {
    new Thread(() -> {
        try {
            Log.d( tag: "ChatService",  msg: "Attempting to establish TCP connection with " + ipAddress + ":" + port);
            socket = new Socket(ipAddress, port); // Establishing the TCP connection
            Log.d( tag: "ChatService",  msg: "TCP connection established with " + ipAddress + ":" + port);

            input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            output = new PrintWriter(socket.getOutputStream(),  autoFlush: true);

            // Set the flag to indicate readiness
            synchronized (this) {
                isReady = true;
                Log.d( tag: "ChatService",  msg: "Setting isReady to true");
                notifyAll();
            }
            Log.d( tag: "ChatService",  msg: "Connection is ready");

            // Get the client's IP address
            String clientAddress = socket.getLocalAddress().getHostAddress();
            Log.d( tag: "ChatService",  msg: "ClientAddress: " + clientAddress);

            // Notify the MainActivity that the connection is established and start ChatActivity
            mainHandler.post(() -> {
                Intent intent = new Intent(context, ChatActivity.class);
                intent.putExtra( name: "ipAddress", clientAddress);
                intent.putExtra( name: "port", port);
                Log.d( tag: "ChatService",  msg: "Starting ChatActivity with IP: " + clientAddress + " and Port: " + port);
                context.startActivity(intent);
            });

            // Start listening for messages
            listenForMessages();
        } catch (IOException e) {
            Log.e( tag: "ChatService",  msg: "Error establishing TCP connection", e);
            e.printStackTrace();
            synchronized (this) {
                isReady = false;
                notifyAll();
            }
        }
    }).start();
}
```

The *start* method initiates a new thread to handle the connection setup on line 37. A socket is then created on line 40 to connect to the server using the provided IP address and port. Input and output streams are set up for communication with the server on line 43 and 44. The service retrieves the server's address and starts the *ChatActivity* passing the server's IP and port as extras (line 59 to 64). On line 68 the method invokes the *listenForMessages* method to continuously listen for incoming messages from the server.

6.4.7 Sending messages between two nodes



In *ChatActivity*, when the *sendButton* is clicked, the text from the messageInput field is extracted, a timestamp is generated, and a *Message* object is created. This message is logged and sent using the chatService's *sendMessage* method, then appended to the chat view with a label and timestamp, and the input field is cleared. In *ChatService*, the *sendMessage* method logs the send request, starts a new thread to handle the message sending, waits for readiness, converts the message to JSON (through the library GSON), sends it over the TCP connection, flushes the output stream, and logs the sent message.

ChatActivity class

```
48          sendButton.setOnClickListener(v -> {
49              String messageText = messageInput.getText().toString();
50              String timestamp = String.valueOf(System.currentTimeMillis());
51              Message message = new Message(messageText, timestamp, ipAddress);
52
53              Log.d( tag: "ChatActivity",  msg: "Sending message: " + message.getMessage());
54              chatService.sendMessage(message);
55              chatMessages.append("Me: " + message.getMessage() + " [" + message.getTimestamp() + "]\n");
56              messageInput.setText("");
57          });
58      }
```

The *sendButton* is configured with an *OnclickListener* that defines what happens, when the button is clicked. On line 49 the text from the *messageInput* fields is extracted and stored in the *messageText* variable. Afterwards a timestamp is generated using the current system time. The *message* object is created of type *Message* on line 51 with the extracted message text, the generated timestamp and the IP address. On line 54 the *sendMessage* method of the *chatService* is called to send the message. This is explained further down. On line 55 and 56, the message is appended to the *chatMessage* view with a label *Me* and the timestamp. Finally, the *messageInput* field is cleared.

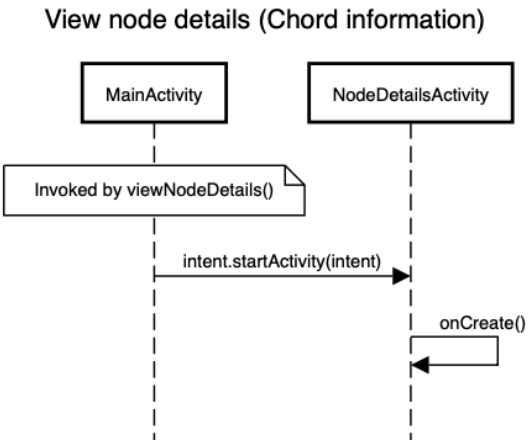
ChatService class

```
        1 usage   jatindershub
97      public void sendMessage(Message message) {
98          Log.d( tag: "ChatService",  msg: "sendMessage called");
99          new Thread(() -> {
100             Log.d( tag: "ChatService",  msg: "sendMessage thread started");
101             waitForReady();
102             Log.d( tag: "ChatService",  msg: "waitForReady completed");
103             if (output != null) {
104                 String messageJson = gson.toJson(message);
105                 output.println(messageJson); // Send message over TCP connection
106                 output.flush();
107                 Log.d( tag: "ChatService",  msg: "Sent message: " + messageJson);
108             } else {
109                 Log.e( tag: "ChatService",  msg: "Output stream is null, message not sent");
110             }
111         }).start();
112     }
```

This method sends a message over the established TCP connection, by starting a new thread to handle the message sending process to avoid blocking the main thread. The *WaitForReady*

method ensures that the service is ready before attempting to send the message (line 101). The message is converted to JSON and sent over the TCP connection using the output stream on line 104 to 106.

6.4.8 View node details (Chord information)



In *MainActivity*, a click listener is set for the *viewDetailsButton* to invoke the *viewNodeDetails* method. This method creates an intent to start *NodeDetailsActivity*, passing the node's ID, predecessor's ID, successor's ID, and finger table as extras, and then starts the activity. In *NodeDetailsActivity*, the onCreate method initializes the UI, retrieves the intent extras, and sets the values to the corresponding UI elements. It also dynamically creates and adds *TextView* elements for the finger table entries and sets up a button click listener to toggle the finger table's visibility.

***MainActivity* class**

```
103             // Other button click listener
                joinNetworkButton.setOnClickListener(v -> new Thread(this::joinNetwork).start());
105             leaveNetworkButton.setOnClickListener(v -> new Thread(this::leaveNetwork).start());
106             viewDetailsButton.setOnClickListener(v -> viewNodeDetails());
107         }
```

The code on line 106 sets up a click listener for the *viewDetailsButton* declared inside of the class of type *Button*. When the button is clicked, it triggers the method *viewNodeDetails*.

```
    1 usage  jatindershub
210 private void viewNodeDetails() {
211     Intent intent = new Intent( packageContext: MainActivity.this, NodeDetailsActivity.class);
212     intent.putExtra( name: "nodeId", node.getNodeId().toString());
213     intent.putExtra( name: "predecessorId", node.getPredecessor() != null ? node.getPredecessor().getNodeId().toString() : "None");
214     intent.putExtra( name: "successorId", node.getSuccessor() != null ? node.getSuccessor().getNodeId().toString() : "None");
215     intent.putExtra( name: "fingerTable", node.getFingerTableAsStringArray());
216     startActivity(intent);
217 }
218
```

The *viewNodeDetails* method creates an intent to start the *NodeDetailsActivity*. It adds extra information to the intent, including the node's ID, predecessor's ID, successor's ID and the

finger table, converting these to string representations where necessary. Finally, the method starts the *NodeDetailsActivity* with the provided intent.

***NodeDetailsActivity* class**

```
23      @Override
24      protected void onCreate(Bundle savedInstanceState) {
25          super.onCreate(savedInstanceState);
26          setContentView(R.layout.activity_node_details);
```

The *onCreate* method is called when *NodeDetailsActivity* is created. It sets the content view to *activity_node_details* on line 26.

```
28          nodeIdTextView = findViewById(R.id.nodeIdTextView);
29          predecessorIdTextView = findViewById(R.id.predecessorIdTextView);
30          successorIdTextView = findViewById(R.id.successorIdTextView);
31          fingerTableLayout = findViewById(R.id.fingerTableLayout);
32          fingerTableScrollView = findViewById(R.id.fingerTableScrollView);
33          toggleFingerTableButton = findViewById(R.id.toggleFingerTableButton);
```

on line 28 to 33 the UI elements get initialized by finding them by their IDs.

```
35          Intent intent = getIntent();
36          String nodeId = intent.getStringExtra( name: "nodeId");
37          String predecessorId = intent.getStringExtra( name: "predecessorId");
38          String successorId = intent.getStringExtra( name: "successorId");
39          String[] fingerTable = intent.getStringArrayExtra( name: "fingerTable");
40
41          nodeIdTextView.setText(nodeId);
42          predecessorIdTextView.setText(predecessorId);
43          successorIdTextView.setText(successorId);
```

It retrieves the intent that started the activity and extracts the node ID, predecessor ID, successor ID and finger table from the intent extra (line 35 to 39). These values are then set to the respective *TextView* elements in the activity (line 41 to 43).

```
45          for (String entry : fingerTable) {
46              TextView fingerEntryView = new TextView( context: this);
47              fingerEntryView.setText(entry);
48              fingerTableLayout.addView(fingerEntryView);
49          }
```

Regarding the finger table, it dynamically creates *TextView* elements for each entry in the finger table and adds them to the *fingerTableLayout*.

```
51          toggleFingerTableButton.setOnClickListener(new View.OnClickListener() {
                jatindershub
52              @Override
53              public void onClick(View v) {
54                  if (fingerTableScrollView.getVisibility() == View.GONE) {
55                      fingerTableScrollView.setVisibility(View.VISIBLE);
56                      toggleFingerTableButton.setText("Hide Finger Table");
57                  } else {
58                      fingerTableScrollView.setVisibility(View.GONE);
59                      toggleFingerTableButton.setText("Show Finger Table");
60                  }
61              }
62          });
63      }
```

Finally, it sets up a click listener for the *toggleFingerTableButton*, which toggles the visibility of the *fingerTableScrollView* and updates the button text accordingly.

6.5 User interface and interaction

This section provides an explanation of the user interface (UI) components of our application and how they interact with each other. We present an overview of the user interface, highlighting key elements. Through concrete code examples, we demonstrate how event handling is implemented to manage user interactions, and how the UI is updated in response to these events.

6.5.1 Overview of the user interface

The user interface component handles all user interactions and displays necessary information. It is composed of multiple activities, each serving a specific purpose within the application. For instance, *MainActivity* manages the main screen, *ChatActivity* handles the chat interface, and *NodeDetailsActivity* displays detailed information about individual nodes. These activities interact with the underlying services and data model to present real-time information and respond to user action.

The UI is built using Androids XML-based layout system. The files *activity_main*, *activity_chat*, *activity_node_details* and *node_item* define the structure and appearance of the various screens. These layouts are linked to their corresponding activities, which handle the logic and interactions, and which is explained in the section Core features.

The application consists of a vertical layout that stacks its child views below the other. Each activity layout file specifies the user interface for different sections of the application, ensuring a clean and consistent user experience.

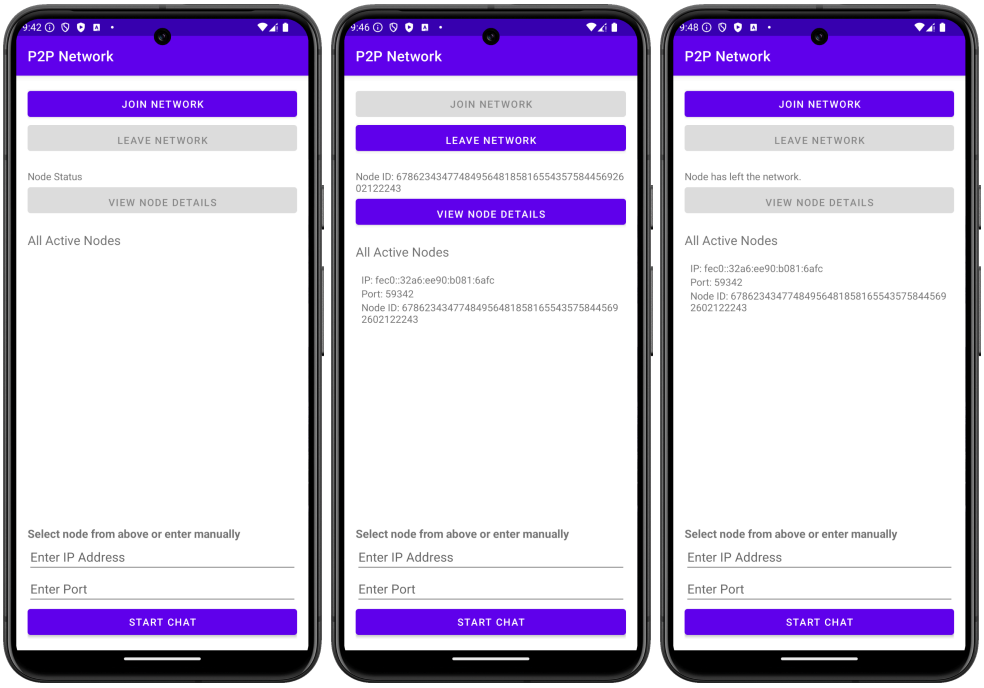
6.5.2 MainActivity layout (activity_main.xml)

This layout serves as the main entry point of the application. It includes a toolbar, and a *RecyclerView* for displaying a list of nodes. The toolbar provides navigation and interaction options, and the *RecyclerView* displays a list of nodes dynamically.

Upon launching the application, users are presented with a straightforward user interface. The interface includes the following components from top to bottom (refer to the left image below for visual context):

- “Join Network” Button: Enabled by default, this button allows users to join the network by clicking on it.
- “Leave Network” Button: Grayed out and disabled, this button indicates that leaving the network is not currently possible.
- “Node Status” Text Field: Displaying the standard label “Node status,” this field provides information about the current status of the node.
- “View Node Details” Button: Disabled and grayed out, this button does not allow users to access node-specific details.
- “All Active Nodes” Section: Initially empty, this section lists all active nodes in the network.

- Bottom Form: This form includes the following elements:
    - Guiding Text: Instructs users to either select a node from the list above or manually enter an IP address and port.
    - Input Fields: Two input fields with placeholders for entering an IP address and port.
    - “Start Chat” Button: Initiates a chat session.



When the user clicks the “Join Network” button (see the picture in the middle for visual context) it transitions from an enabled state to a disabled state. The *joinNetwork* method in the *MainActivity* class updates the UI to reflect the node’s status on the main thread. We describe the logic behind the *joinNetwork* method in the section Core features.

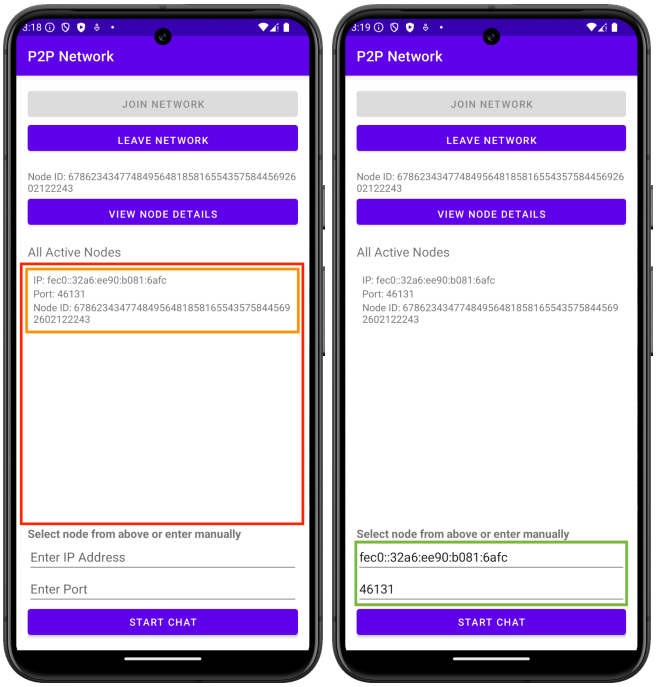
```
249                runOnUiThread(() -> {
250                    nodeStatus.setText("Node ID: " + node.getNodeId());
251                    viewDetailsButton.setEnabled(true);
252                    leaveNetworkButton.setEnabled(true);
253                    joinNetworkButton.setEnabled(false);
254                });
```

This action, in turn, enables both the “Leave Network” and “View Node Details” buttons. The reason behind this behavior is that the node has successfully joined the network and become an integral part of the Chord ring. Consequently, the nodes’ information is displayed within the RecycleView, appearing under the “All Active Nodes” section. Every object from the RecycleView is an instance of the “node_item.xml”, which contains relevant details such as IP address, port, and node identifier. These details are sourced from the backend, as

explained in the “Core Features” section. Additionally, the status of the node is updated with the corresponding device ID information.
When the user clicks the “Leave Network” button, the application returns to the same state as when it was initially launched (see picture above to the right).

Initially, we intended to display a join button on the right side of each *nodeInfo* object in the interface, allowing the user to connect directly to another node with a single click. However, due to time constraints and the need to prioritize a minimum viable product (MVP), we opted for a simpler solution: two input fields and a button to initiate the TCP connection between two nodes.

To enhance usability, we introduced a workaround that enables the user to click on a *nodeInfo* object. This action automatically populated the IP address and port number into the input fields, so the user only needed to click the button to initiate the TCP connection with the selected node. This streamlined process eliminates the need for manual data entry, making the connection process more efficient and user friendly.



The pictures above represent the UI after a node has joined the network.
The red square (left) represents the *RecyclerView* and the orange square (left) represents a *nodeInfo* object, which is inside of the *RecyclerView*. When an object in the *RecyclerView* is clicked on, the IP address and port number from the clicked object are retrieved and populated into the corresponding views *ipAddressInput* and *portInput*, which is seen in the green square (right).

When the user clicks on the *nodeInfo* object, the following code gets called inside of the *onCreate* method in *MainActivity*.

```
104        nodesAdapter.setOnItemClickListener(nodeInfo -> {
105            ipAddressInput.setText(nodeInfo.getIp().getHostAddress());
106            portInput.setText(String.valueOf(nodeInfo.getPort()));
107        });
```

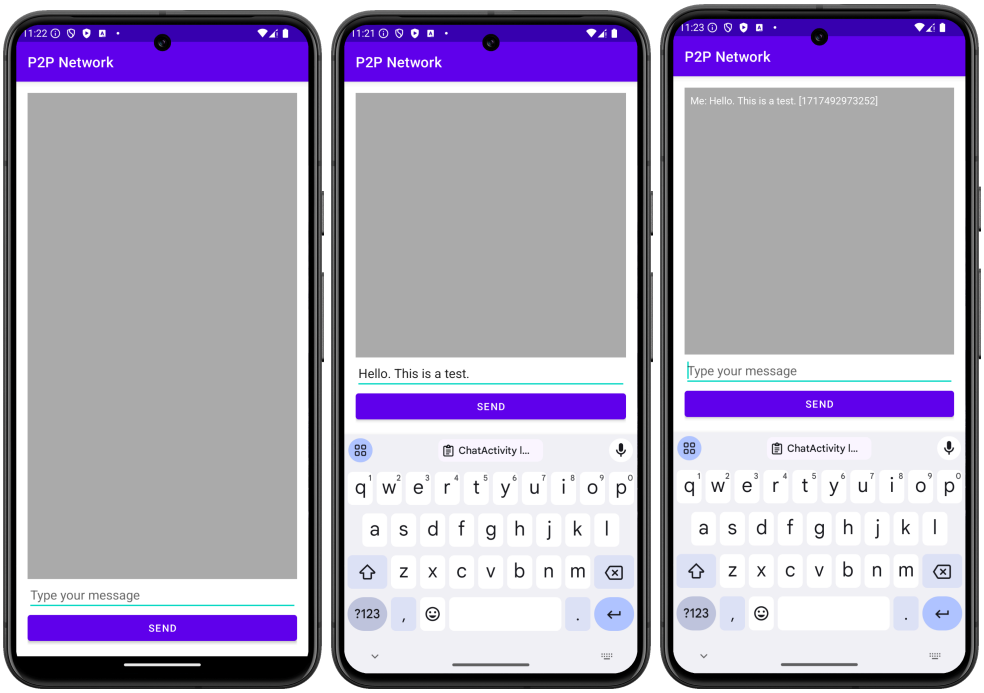
On line 104 there is an item click listener on the *nodesAdapter*. The *setOnItemClickListener* method is a custom method defined in the *NodesAdapter* class, which handles click events on items in the *RecyclerView*.
The lambda expression *nodeInfo -> {...}* specifies the action to be performed when an item is clicked:

- Line 105: This sets the text of the *ipAddressInput* view to the IP address of the clicked *nodeInfo* object
- Line 106: This sets the text of the *portInput* view to the port number of the clicked *nodeInfo* object

The user can then click on the "Start chat" button, which triggers the activity_chat.xml, which is described down below.

6.5.3 ChatActivity layout (activity_chat.xml)

This layout manages the chat interface, displaying messages in a list (*RecycleView*) and providing an input field for new messages.



When launching the *activity_chat.xml* the user is met with a large area intended to display the chat messages, an input field with the placeholder “Type your message”, and finally an “Send” button (see the left picture above for visual context).

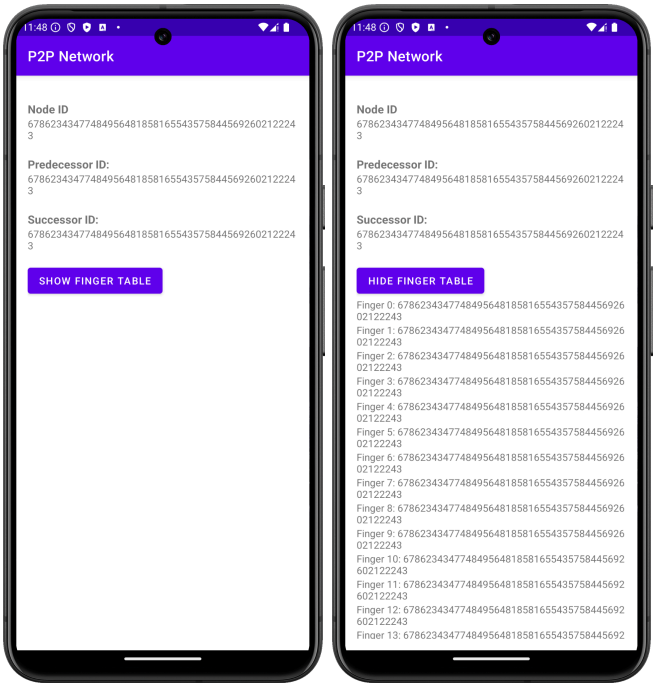
When the user types a message into the input field (see picture from above in the middle for visual context), the *EditText* component captures the user’s input. The *EditText* component has an ID which allows it to be referenced in the *ChatActivity* class.

When the user presses the “Send” button, which also has an ID, the *ChatActivity* captures this event through an *OnClickListener*. This listener is set up in the *ChatActivity* class and triggers the process of sending the message. The message is then displayed as a *ChatMessage* object within the *RecyclerView* (see the picture above to the right).

6.5.4 NodeDetailsActivity layout (activity_node_details.xml)

This layout displayed detailed information about a specific node, including its node ID, predecessor and successor.

Then the user selects “View node details” within the *MainActivity* class, the information depicted in the accompanying picture below (left) becomes visible. Initially, the *Node ID*, *Predecessor ID* and *Successor ID* are displayed. Additionally, the “Show finger table” button is initially collapsed by default for improved usability due to its extensive content. However, users can expand this button by clicking on it, revealing a scrollable list of finger tables (see picture below to the right).



6.6 Unit testing

In our project, and in all software development for that matter, ensuring that individual components of a system performs as expected is important. To achieve this, we perform unit testing on our code, which involves testing chunks, or units, of code, to verify the correctness of the code. Our approach to this has been a sort of white-box testing with elements from black-box testing, which can be looked at as gray-box testing.

Gray-box testing merges elements of both black-box and white-box testing. So when we conduct gray-box testing, we have partial knowledge implemented in the tests. This approach allows us to write tests that are more informed than black-box testing, which treat the application as a "black box" without knowledge of the code, and yet not as deeply involved as classic white-box tests, which require full knowledge of the code structure to be implemented in the tests.

For instance, the tests for the creation of a ChordNode object validate the components of the object after the creation. This includes validations such as checking the IP address, nodeId, and port. This approach ensures that the fields are correctly initialized, showing an understanding of the code while focusing on the output. Below, a picture of one of the ChordNode tests is seen:

```
90      @Test
91      public void testChordNode_IpNodeIdPort() {
92          System.out.println("\ntestChordNode_IpNodeIdPort start");
93
94
95          ChordNode chordNode = new ChordNode(ip, nodeId, dynamicPort);
96          System.out.println("0");
97
98          assertNotNull(chordNode);
99          System.out.println("1");
100         assertEquals(ip, chordNode.getIp());
101         System.out.println("2");
102         assertEquals(dynamicPort, chordNode.getDynamicPort());
103         System.out.println("3");
104         assertEquals(nodeId, chordNode.getNodeId());
105         System.out.println("4");
106         assertNull(chordNode.getPredecessor());
107         System.out.println("5");
108
109         System.out.println("testChordNode_IpNodeIdPort passed");
110     }
```

By deploying unit testing throughout our project, we have been able to detect errors in early stages of development. Relating to that, we have been able to isolate errors to specific

methods. By testing, for instance, the “getAvailablePort”-method, we have been able to see if it returned a correct free port, and if we could bind it to a socket. Therefore, we have been able to isolate errors relating to this method to very specific parts of the code. All in all, this has led us to better code quality in specific methods, which are crucial to multiple parts of the code.

6.7 Unresolved issues and limitations

During the development of our application, we identified several issues that we were unable to resolve before the project submission. These known issues are outlined below and should be taken into consideration for future iterations of the application. The inclusion of unresolved issues and limitations is an essential component of our documentation, ensuring transparency.

The issues identified have been both regarding the functionality of our code and the tools we had access to during the development.

#	Unresolved issue	Impact	Potential solution
1	Node not being removed from the *RecyclerView* (*All Active Nodes*) when clicking the leave button.	The *RecyclerView* still displays nodes that have left the network, leading to inaccurate representation of active nodes.	Ensure the *updateNodeList* method is properly called after sending the *LEAVE* message, and verify that the method *handleReceivedMessage* implemented inside of the class *MulticastService* correctly updates the node list by removing the node and calling *nodeListUpdater* to refresh the *RecyclerView.*
2	The *sendMessage* method in the *ChatService* class gets stuck at “Waiting for readiness” because the *isReady* flag is not being set to *true*.	Users experience delays or indefinite blocking when attempting to send messages, leading to frustration and communication failures. The core functionality of the chat application is compromised as messages are not transmitted, but only displayed in the UI for one node.	Ensure that the *isReady* flag is set to *true* after the TCP connection and streams are initialized. Use proper synchronization with *notifyAll* to wake up waiting threads. Add detailed logging to trace the flow and identify where it might be getting stuck.

3	Networking issues on emulators inside of Android Studio.	Inability to test network functionalities accurately, leading to potential undetected bugs and unverified features.	Test on physical devices to ensure accurate networking behavior, or configure the emulator to better handle network operations. Investigate and apply emulator settings to simulate real-world conditions more accurately.

6.8 Potential attacks

Throughout our design and implementation process, we have considered potential attacks. We have identified both some of the vulnerabilities we have addressed and some we have not fully resolved. Even for the unresolved issues, we have attempted to lessen their impact or at least recognized them as risks.

A fundamental but unavoidable vulnerability in data transfer is the risk of man-in-the-middle (MITM) attacks. However, this risk is mitigated by well-established measures such as data encryption. We plan to use AES encryption, which significantly enhances data security and makes it difficult to manipulate. However, at this point in time, we have not yet implemented the encryption. Nonetheless, a key weakness remains: AES is symmetric encryption, meaning the key must be shared between sender and receiver. This creates an opportunity for a MITM attacker to intercept the key and gain access to the conversation, allowing them to compromise the data.

To mitigate this risk and enhance overall security, we had implemented a unique approach for a chat application: disallowing permanent profiles. This decision, while challenging from a design and coding perspective, reduces security risks like MITM attacks. By creating a new profile for each chat session, we minimize the impact of intercepted encryption keys. This approach also mitigates attacks, such as phishing attacks and other social engineering tactics aimed at accessing user profiles. In distributed peer-to-peer networks, losing a profile or altering data can cause significant issues since there is no central control node. Our approach means there is less lasting damage from such attacks, although the threat remains.

Currently, our application is somewhat vulnerable to Distributed Denial of Service (DDOS) attacks. If the application expands, this vulnerability could become a major issue, as we lack specific security measures to mitigate DDOS attacks. As the application grows to use the global internet, instead of just the local network, there is no way of avoiding the requirement of one or more anchor nodes for accessing the network, increasing the consequences of DDOS attacks. Anchor nodes or specific users could be overwhelmed by volume attacks, causing users to disconnect. This is a minor issue now, but it will become more significant when connection routing between nodes is more crucial, making DDOS attacks more feasible and disruptive. A successful DDOS attack could disconnect users, making it difficult for

them to reconnect, especially if they only know each other through one node. Also, a DDOS attack against an anchor node could prohibit a new node from accessing the network.

Lastly, we have considered the threat of malware propagation or injection. This involves malicious code or links being uploaded to a node and spreading to users' devices when clicked. While this is not an immediate threat, it could become one if the application expands and adds new functionalities. To address this, we could limit what can be shared within the application, though this would restrict its functionality. Another approach could be to implement warnings about downloading content, shifting responsibility to the user while maintaining product flexibility.

7 Discussion

In this discussion we will go over the lessons we have learned, what we can continue improving in this project, and what its strength and weakness is its implementation. It will start by summing up all the strengths and weaknesses we have found in the analysis. Thereafter it will smoothly transition into what we can improve on in the future of the project based on those strengths and weaknesses and other things that have been found lacking in the implementation. Finally, there will be a reflection over what was learned throughout this project both in coding and organizationally in preparing and organizing the workflow of the development.

7.1 Strengths and weaknesses of our implementation

The implementation of the Chord connection protocol is both a significant strength and weakness of this project. On the positive side, having Chord in place will make it much easier to scale the system to support thousands of users on an anchor node if the project ever expands beyond local networks. This is because Chord streamlines the process of sorting and indexing users, making it simpler to handle the constant addition and removal of users. However, the Chord implementation is currently missing a crucial element: the lookup algorithm. While not essential for establishing a Chord connection, the lookup algorithm is necessary for searching for users when it becomes impractical for a single node to have complete knowledge of all other nodes. Additionally, Chord is notoriously difficult to implement, so a substantial amount of time and effort is required to maintain its functionality as new features are added to the application.

As previously mentioned, the application is currently limited to local networks, which is a major weakness given its original vision. This restriction makes it less functional for the average user, who typically does not share a network with the people they wish to communicate with. There are some exceptions, such as an intranet for a university, where the added security of restricting access to network members might be desirable. However, transforming the application to suit this use case would require significant work.

The core functionality for enabling user communication relies on UDP connections for adding users to the Chord network and TCP connections for facilitating individual chats. This functionality is intuitive and user-friendly, requiring no advanced technical knowledge. However, there are some weaknesses to consider. The UDP connection is vulnerable due to its lack of protections, with IP addresses and ports openly displayed.

Some less obvious weaknesses in our implementation include the lack of resilience in our nodes. As discussed earlier, this vulnerability could be exploited by attackers to remove nodes from the network. Furthermore, we have not yet determined how to connect with nodes outside a local network, so if a node leaves the local network, they will have no way to rejoin without resorting to communication methods outside the application.

Despite the various implementation shortcomings from a cybersecurity perspective, one of the central tenets of this application is its strong commitment to privacy. The application is designed to allow users to communicate without revealing any personal information and without storing data in any way that could potentially be accessed by threats. The application intentionally avoids handling sensitive user data, which means it requires less stringent security measures and central oversight compared to more traditional chat platforms.

Finally, we implement JSON formatting in the chat messages. This allows us to send a bigger volume of information with each message, which is easily read by the receiver, such as not the message itself but also information like timestamp, recipient, sender, and other information that might be useful. This is also a way to create a base for further expansion of the messaging format. Even with the use of JSON formatting, we are still limited to only sending plain text and not sharing images or files. Even though this limits the use cases of the application, it makes risks of malware propagation and injection much lesser.

7.2 What can be improved on

This section outlines the improvements necessary before publicly releasing the application, rather than the next steps in development.

7.2.1 Integrating encryption and hashing

End-to-end encryption (E2EE) is essential for ensuring that only the communicating users can read the messages. This is critical for preventing unauthorized access, protecting privacy and safeguarding sensitive information from interception by malicious actors. Without encryption, data/messages transmitted over the network can be vulnerable to various attacks, such as man-in-the-middle attacks, where an attacker can intercept and potentially alter the messages being exchanged.

In our application nodes discover each other through UDP and establish a direct TCP connection for communication. During this process, there is a significant risk of man-in-the-middle attacks. Implementing E2EE mitigated this risk by ensuring that even if the communication is intercepted, the attacker cannot decipher the contents of the messages without the encryption keys.

For the encryption implementation, we would use the Advanced Encryption Standard (AES) for encrypting the messages and RSA for secure key exchange. Additionally, we would use SHA-256 for hashing to ensure data integrity. In our *sendMessage* method we would firstly ensure that each node can generate and share encryption keys securely.

An approach for integrating encryption and hashing mechanisms in our existing code, would be in the method *sendMessage* in the *ChatActivity* class.

The *initializeKeys* method would be responsible for generating a pair of RSA keys for each node. The public key would be shared with the recipient, while the private key would remain secret. Additionally, an AES key would be generated, which would be used for encrypting the messages.

In the *sendMessage* method, the message would be encrypted using the AES key. This encryption would ensure that only the intended recipient, who has the appropriate decryption key, would be able to read the message. After encrypting the message, a hash of the original message would be created using the SHA-256 algorithm. This hash would serve as a digital fingerprint of the message, allowing the recipient to verify that the message has not been altered during transmission. The encrypted message and the hash would get concatenated and sent together over the network.

Upon receiving a message, the method *receiveMessage* would handle the reception process. The received message would be split into the encrypted message and the hash. The encrypted message would then be decrypted using the AES key. To ensure the integrity of the message, a hash of the decrypted message would be computed and compared with the received hash. If the hashes match, it would confirm that the message has not been tampered with, and the decrypted message would then get displayed to the user.

7.2.2 Expanding beyond local networks

After ensuring the encryption algorithm and data exchange formats are secure, the first priority would be improving the lookup algorithm for the Chord structure. As previously mentioned, a robust lookup algorithm is crucial for the Chord network to realize its full scalability potential. While not essential for establishing the ring, it greatly simplifies searches within the Chord network, allowing the algorithm to demonstrate its true strength. Without this, the Chord structure will be hindered and unable to scale effectively. Therefore, implementing the lookup algorithm is necessary before proceeding to the next application stage.

The next objective would be enabling the application to expand beyond its current local network user base and connect to the wider internet. Currently, users are limited to connecting only within a restricted network, contrary to the chat service's intended purpose of allowing connections across distances. Internet access would also reduce dependence on local networks, which control anchor node placement and may block outside traffic. This is particularly important given that many chat applications are used on-the-go, and users are often not consistently on the same network. However, as noted in the potential attacks and application strengths/weaknesses sections, internet connectivity introduces new vulnerabilities and connection challenges. Mitigations like temporary user profiles have been considered to address these issues.

7.2.3 Improved usability

To improve the usability of our application, we have primarily considered the principles from Shneiderman and Plaisant "Designing the User Interface: Strategies for Effective Human-Computer Interaction" (2010). One critical enhancement is the implementation of usernames instead of displaying IP addresses. This change makes the application more user-friendly and secure, as users no longer need to read or write IP addresses, which are tedious and error-prone.

Additionally, to protect users' privacy and security, we would hash IP addresses and use these hashed values as node IDs, ensuring that real IP addresses are not broadcasted. This approach aligns with the principle of preventing errors by designing the system to avoid potential security risks.

We would also provide clear and informative feedback messages, ensuring users are aware of the status of their actions. For instance, when a message is sent, users would receive a confirmation that the message was delivered successfully. This practice aligns with the principle of offering informative feedback and designing dialogues that yield closure. Another example is notifying users when their connection is established or lost, providing a clear understanding of the system's status (Shneiderman & Plaisant, 2010).

Moreover, we would allow users to reverse their actions, such as deleting or editing a sent message, which supports the principle of permitting easy reversal of actions. This feature ensures users feel in control and reduces frustration from irreversible mistakes. For instance, if a user mistakenly sends a message to the wrong person, they could easily retract it and send it to the correct recipient.

To support the internal locus of control, the application would be designed so that users feel they are in command. By ensuring actions are straightforward and providing immediate feedback, users would feel more confident in using the application. For instance, users could easily change their settings or update their status, and these changes would be immediately reflected in the system.

Finally, to reduce the short-term memory load, the application would be designed to minimize the amount of information users need to remember. Using usernames instead of IP addresses, providing clear and consistent feedback, and ensuring the interface is intuitive and easy to navigate could help achieve this. For instance, users could see a list of their contacts with easily recognizable usernames instead of remembering complex IP addresses or node IDs.

By incorporating these usability improvements, we aim to make our application not only more secure but also more accessible and enjoyable for users. These enhancements would ensure a smoother, more intuitive user experience, thereby increasing user satisfaction and engagement (Shneiderman & Plaisant, 2010).

7.3 What have we learned

One of the key takeaways from this project has been gaining a deep understanding of the Chord protocol's structure and the challenges of implementing it. While the implementation has driven much of our project architecture, it has not been without its difficulties. We have learned valuable lessons about how to manage our time effectively and when to prioritize Chord implementation. While Chord will ultimately enable massive scalability, achieving this has required a substantial investment of time and effort to ensure it works seamlessly with every new feature. Given the current implementation is confined to a local network with a limited number of users, the full power of Chord remains untapped, and we have yet to determine how to extend it to a real-world setting.

As discussed in the potential attacks section of the report, the resilience of peer-to-peer (P2P) networks is inherently fragile. While the lack of a central server or control service makes them less vulnerable to certain types of attacks, it also deprives developers of the tools typically used to maintain standards, protect users, and ensure node security. To mitigate this, we've had to implement measures like making users temporary and facilitating easy joining and leaving. This tradeoff is a crucial consideration when building P2P networks.

This experience has underscored the paramount importance of security in protecting user data and the complexity of securing it from all angles. In this project, the most feasible and aligned solution was to prioritize user privacy and minimize data storage. By design, the application avoids handling sensitive user information, which inherently reduces the attack surface and the risk of data breaches. There are no user profiles to steal through phishing, and tracking conversations through man-in-the-middle (MITM) attacks becomes less long term impactful since users can recreate their profiles as needed.

The most critical, broad, and applicable lesson we have learned throughout this project is the importance of a structured workflow. We began by creating a Kanban board and schedule on Trello, which greatly facilitated communication and task management. As this proved successful, we placed greater emphasis on refining our organizational processes. We augmented our schedule with milestones, enabling team members to better synchronize their projects and workflows. The introduction of sprints further enhanced coordination, ensuring each week yielded significant progress and kept everyone aligned. This structured approach has been vital in maintaining smooth development momentum, even after implementing major features.

8 Conclusion

In conclusion, this study has illuminated several challenges and strengths of P2P networks, insights that will inform future efforts in this domain and analogous technological areas. These findings underpin all aspects of this research, from the creation of the Chord network to the implementation of the chat feature. To provide a comprehensive summary of this article, we will revisit the research questions and describe the contributions each has made to our overall understanding.

Our first research question was “**How can one build a peer-to-peer communication network?**”. To accomplish this, we learned that a significant amount of architectural planning was required, as establishing connections between nodes is not a straightforward task. We also gained a better understanding of the essential factors needed to make a Peer-to-Peer (P2P) network function effectively, particularly in terms of facilitating communication.

The second research question, "**How can a node efficiently find and connect to other nodes in a serverless peer-to-peer network?**" logically emanates from the first question. Indeed, upon initialization, nodes must locate each other to facilitate meaningful interaction within the network. To enable this crucial connection, we leveraged the UDP, which permits constant broadcasting. This allowed nodes to discover each other within a local network, thereby establishing initial connections. Subsequently, to support the peer-to-peer chat functionality inherent in a chat application, a TCP connection was implemented. TCP provides greater security and reliability, enabling more private and robust conversation between nodes.

This leads to the third research question: "**What challenges arise from implementing a peer-to-peer network based on the Chord protocol?**" The reason for this question is that while nodes can initially find each other in a network, this is not a sustainable solution if scalability is desired. To address this, an attempt was made to implement the Chord protocol. However, challenges quickly emerged because Chord is an advanced protocol requiring many moving parts and complex logic at each step. This meant that while a partial implementation of the Chord system was possible, it imposed a heavy burden on resources. If the application were to be further developed, this would remain a significant issue.

In developing a chat application that handles user data, we recognized the importance of enhancing communication security within a peer-to-peer (P2P) network. This led us to research the question, "**What considerations can be taken to improve communication security in a P2P network?**" Through our research, we gained valuable insights into the unique characteristics of P2P networks. While they may not be as resistant to direct attacks as other network types, their decentralized nature makes them resilient, with no single point of failure that could take down the entire network.
We also identified potential risks, such as user data theft and malware, and explored strategies to mitigate them. Importantly, we recognized that P2P networks have an inherent risk factor

that cannot be fully eliminated. Rather than trying to achieve absolute security, we focused on designing the application to work effectively within this reality by incorporating failure modes into its operation.
For instance, the application's ability to quickly onboard new users reduces the impact of denial-of-service (DOS) attacks. Similarly, while man-in-the-middle (MITM) attacks are a serious concern, we have implemented measures to make it difficult for attackers to consistently target specific users without concerted effort. By understanding the unique security challenges and opportunities of P2P networks, we have aimed to create a more secure and resilient chat application for our users.

Our research and development have provided valuable insights into answering our thesis: "**How can you create a serverless peer-to-peer communication network as a chat application based on the Chord protocol?**" We have learned that building a highly resilient P2P network requires careful consideration of its architecture and ongoing maintenance to ensure components continue functioning as intended as new features are added.
While challenges are inherent to this approach, the benefits of distributed, Chord-based network make it a viable choice for a secure and robust chat application. By understanding the unique security considerations and proactively mitigating risks, we can create a communication platform that leverages the strengths of P2P technology while minimizing its vulnerabilities.

9 References

Anderson, D. J. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press.

Desikan, S., & Ramesh, G. (2006). *Software Testing: Principles and Practices*. Oxford University Press.

Grabbe, J. O. (2018). *The DES Algorithm Illustrated*. Private Homepages. Retrieved May 1, 2024, from https://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm

Kniberg, H., & Skarin, M. (2010). *Kanban and Scrum: Making the Most of Both*. C4Media Incorporated.

Kurose, J., & Ross, K. (2018). *Computer Networking: A Top-Down Approach, Global Edition*. Pearson Education.

MacMillan, D. (2018, July 2). *Tech's 'Dirty Secret': The App Developers Sifting Through Your Gmail*. Wall Street Journal. Retrieved June 4, 2024, from https://www.wsj.com/articles/techs-dirty-secret-the-app-developers-sifting-through-your-gmail-1530544442

Mahajan, P., & Sachdeva, A. (n.d.). *A Study of Encryption Algorithms AES, DES and RSA for Security*. Global Journals. Retrieved May 5, 2024, from https://globaljournals.org/GJCST_Volume13/4-A-Study-of-Encryption-Algorithms.pdf

Mahmoud, M. S. (2011). *Decentralized Systems with Design Constraints*. Springer London.

Manifesto for Agile Software Development. (2017, November 9). Manifesto for Agile Software Development. Retrieved May 31, 2024, from https://agilemanifesto.org/iso/en/manifesto.html

MIT Laboratory for Computer Science. (n.d.). *Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service*. cs.Princeton. Retrieved April 10, 2024, from https://www.cs.princeton.edu/courses/archive/spr05/cos598E/bib/dabek-chord.pdf

MIT Laboratory for Computer Science. (n.d.). *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. MIT PDOS. Retrieved April 5, 2024, from https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

Newman, L. H. (2018, December 10). *Google+ Exposed Data of 52.5 Million Users and Will Shut Down in April*. WIRED. Retrieved June 4, 2024, from https://www.wired.com/story/google-plus-bug-52-million-users-data-exposed/

Rivest, R. (2017, 4). *11 Hashing Techniques: A Survey and Taxonomy*. Department of Electrical Engineering and Computer Science. Retrieved April 30, 2024, from https://www.cse.fau.edu/~xqzhu/papers/ACS.Chi.2017.Hashing.pdf

Shneiderman, B., & Plaisant, C. (2010). *Designing the User Interface: Strategies for Effective Human-computer Interaction*. Addison-Wesley.

Sriparasa, S. S. (2013). *JavaScript and JSON Essentials*. Packt Publishing.

Steen, M. v., & Tanenbaum, A. S. (2017). *Distributed Systems*. CreateSpace Independent Publishing Platform.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., & Balakrishnan, H. (n.d.). *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*. MIT PDOS. Retrieved May 5, 2024, from https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf

Tanenbaum, A. S., & Steen, M. v. (2007). *Distributed systems : principles and paradigms*. Pearson Prentice Hall.

Yu, T., & Jajodia, S. (Eds.). (2007). *Secure Data Management in Decentralized Systems. Advances in Information Security.* Springer US.

10 Appendix

Github repositories
First and second iteration:
https://github.com/Deznyz/Serverless-Distributed-Communication-Network
Third iteration: (**main project**): https://github.com/jatindershub/P2P_Network.git