Technology Stack for Decentralized Mobile Services

Matouš Skála

TU Delft

Technology Stack for Decentralized Mobile Services

by

Matouš Skála

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday August 31, 2020 at 3:00 PM.

Student number:	4893964	
Project duration:	November 15, 2019 – August 31, 2020	
Thesis committee:	Dr.ir. J.A. Pouwelse,	TU Delft, supervisor
	Dr. J.S. Rellermeyer,	TU Delft
	Dr. N. Yorke-Smith,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

When I was choosing my thesis topic, I originally came up with an idea of designing a decentralized social network. After realizing how ambitious that goal was, I later decided to focus on more fundamental issues first and create a library that would allow for building any decentralized applications, running purely on an overlay network consisting of smartphones. Rather than reinventing the wheel, I took inspiration from an existing networking library developed at TU Delft over the last decade and created its wire-compatible implementation in Kotlin. Interestingly, in the end, I have even implemented a trivial social network to demonstrate the usage of the library, returning back to the original idea.

I would like to thank my supervisor Johan Pouwelse for an endless stream of fresh ideas and valuable feedback, and to PhD students of the Delft Blockchain Lab for numerous coffee meetings and for serving me as a walking documentation of the existing codebase.

Matouš Skála
Prague, August 2020

Contents

1 Introduction

The Internet was created with the idea that any two computers connected to the shared network should be able to communicate with each other. It has also been built on the principles of decentralization, without any central entity having the power to take the network down. Yet, 50 years later, we live in a world where most of the services are centralized and user data are stored on the servers owned by a few large profit-oriented companies.

In recent years, the idea of decentralization has attracted many in the engineering and research community. Since the introduction of cryptocurrencies in the last decade, there have been many discussions on whether we can decentralize other services, such as social media, or web. With the trend of decentralization, applications are shifting from the client-server model to peer-to-peer, which brings many challenges and calls for a new networking stack.

This thesis proposes and implements a protocol for peer to peer (P2P) communication between any two devices. It is implemented as a Kotlin library which can be used on a desktop, smartphones, tablets, and IoT devices. It can be used to deploy a truly ubiquitous network overlay which is available anytime and everywhere. The protocol allows any two devices to establish a direct connection by taking advantage of NAT traversal techniques to connect peers behind different types of middleboxes. When the Internet connection is not available and peers are located in proximity, the connection can be established using Bluetooth Low Energy.

The robustness of the NAT traversal mechanism has been tested by conducting a connectivity check between devices using the networks of major mobile network operators and home broadband providers in the Netherlands. The mechanism has been shown to be capable of establishing a connection in all tested network conditions.

To demonstrate the usage of the library, a decentralized social network with public feeds and private end-to-end encrypted messaging has been implemented. To get feedback on the APIs and general usability of the library, 4 teams of MSc students have been asked to develop non-trivial distributed applications on top of it.

Compared to the state of the art solutions, the proposed library combines both nearby and Internet connectivity, does not require any central server, works on a variety of devices under challenging network conditions, and is completely open source.

2

Problem Description

Over the last decade, the Internet and online services have become omnipresent in everyday lives of citizens across the world – communication, shopping, transportation, accommodation, or financial services are only some examples of services where most of the economic activity has already shifted to the online world. Most of these services are operated by so-called *Big Tech* [33], the largest and most dominant companies in the information technology industry – Facebook, Amazon, Google to name a few. In fact, 7 most valuable companies in the world are now tech companies. It seems to be nearly impossible to function in a digital world outside of the ecosystems created by the big tech. For instance, we require permission from Google and Apple to publish software for mobile devices. Their monopoly power means no other meaningful method exists to reach billions of smartphone users with newly created apps.

2.1. Antitrust Battle Against Big Tech

About 2 billion users interact with Facebook every day. It maintains 75% social media market share. In a truly free market, they would compete by superior features and providing the best services for their customers to increase consumer welfare. Instead, Facebook has chosen to mislead and exploit consumers and publishers. The evidence discussed in an analysis paper by prof. Fiona M. Scott Morton from Yale University [19] shows that Facebook has for a decade "engaged in a long-term, integrated, anti-competitive strategy of half-truths about its privacy policies, exclusionary API manipulation, and anti-competitive acquisitions of nascent competitors". It has acquired various competitors such as Instagram (a social network emphasizing photo sharing) and WhatsApp (a widely popular messaging app) for a hefty premium, just to secure its dominance on the market. Because the market is characterized by a strong network effect, it is unlikely to change without a significant regulatory intervention or consumer uprising.

Over the last months, Departement of Justice, Federal Trade Commission, and the European Union have all started investigations of Google, Facebook, Apple, and Amazon over potential illegal anticompetitive behavior. Very recently, in July 2020, hundreds of large businesses united during the *Stop Hate for Profit*[1] campaign and paused all advertising on Facebook platforms for a month to signal disagreement with company's practices and policies. In August 2020, Epic Games filed an antitrust lawsuit [32] against Apple after it removed its game from App Store and revoked all their developer certificates for breach of rules.

[1] https://www.stophateforprofit.org/

2.2. The Threat of Super Apps

Meanwhile, a new paradigm in smartphone software engineering has emerged from the East. In 2017, Tencent introduced the concept of a super app, an app that bundles an app store and allows to extend its functionality with mini-apps. In a short time, 1 million apps have been developed for WeChat, the most popular communication platform in China, and it has attracted 200 million daily users [17]. WeChat now acts as the ultimate app for a Chinese citizen which bundles online messaging, social media, marketplaces, ride-hailing, food delivery, and other services. Last but not least, it offers financial services. In 2018, over 83% of payments in China were made using mobile payment methods, with WeChat Pay and Alipay having the major share [6]. While having a complete social graph and most of the real-world interactions in a single app is useful for providing a good user experience, it also makes it the perfect spy tool. It acts as an unprecedented vault of social, financial, personal, and behavioral data. WeChat is operated by Tencent, a big tech company which is essentially owned by the Chinese government. There has been strong evidence that the app is posing significant privacy risks and is being used to spy on its users [14]. For these reasons and censorship concerns, WeChat is posed to be banned in the USA later this year [36].

Inspired by the trend of super apps, other tech leaders seem to follow the trend. Uber has announced its works on a super app which is posed to become "an operating system for everyday life". It will combine food and grocery delivery, as well as various forms of transportation services [15]. The Facebook app already combines not only social media, but also a marketplace, and soon will include mobile payments with the Project Libra[2] which aims to become a new global payment system. Providing integrated experiences and having access to even more personal data ultimately makes corporations more powerful than sovereign nations.

2.3. End-to-End Principle Challenged

While the Internet is built on the principles of decentralization from the ground up, we cannot say so about the Web and other services built on top of it. Users and their data are kept hostage to big tech companies. Big tech has created a monopoly at the business level, but there are fundamental issues at the network intrastructure level as well, preventing us from shifting towards decentralized system architecture.

The architecture of the Internet has emerged in an evolutionary process rather than from a carefully designed grand plan. There is no central entity in charge of architectural decisions or anyone with the ability to turn it off. Most decisions related to protocol and system design have for a long time followed the *end-to-end principle*, which states: "The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible." [27]

It implies that the job of the network should be merely to deliver datagrams as efficiently as possible, and the rest of the responsibilities should be pushed to the end points of the communication system. As every application has different requirements, providing additional features on the Internet layer would turn some applications not using those features less efficient. It is also cheaper to upgrade the end points to add new capabilities rather than replace the network infrastructure.

The end-to-end principle has been challenged by the introduction of *middleboxes*, intermediary nodes in the network that perform functions different from functions of a regular IP router. In [4], an extensive catalogue of 22 different middlebox classes has been analyzed. Network address translators (NATs), packet classifiers, IP firewalls, application layer gate-

[2] https://libra.org

ways, or proxies are just a few examples of components hidden in the network. While most of them try to be transparent, the behavior of certain middleboxes can severely impact the end-to-end performance, either by delaying packet retransmission or aborting user sessions entirely in case they crash. Some of them perform advanced logic on different protocol layers and keep a hard state, which violates the end-to-end principle stating that the network should be kept as simple as possible.

In the Internet Protocol, each computer gets assigned an address which is subsequently used for packet routing. IPv4 uses a 32-byte address space, which is not enough to uniquely identify all devices on the planet. To deal with IPv4 address exhaustion, internet providers were forced to deploy different types of NATs which allow a single address to be shared across multiple devices.

Traditionally, each consumer would have a NAT implemented in the router located at the edge of the network. It would serve as a gateway between the local area network (LAN) and the wide area network (WAN), the Internet. This type of NAT usually maps all addresses inside LAN to a single external WAN address. It also provides additional features such as firewall, which only allows incoming traffic on explicitly open ports, or on whose the client initiated the communication first. However, to further conserve the address space, and facilitate transitions to IPv6 with backwards compatibility of IPv4, it has become a trend among ISPs to implement a NAT in their infrastructure. This topology is called a *carrier-grade NAT* (CGN). CGN is by definition managed by the network operator and the customer does not have any control over it. CGN usually implements a complex functionality to ensure reliability for thousands of subscribers and compliance with legal regulations, which again violates the end-to-end principle. Inability to manage open ports breaks the communication model of many peer-to-peer applications which usually have to reside to using a proxy server for relaying communication.

To mitigate this issue, *Port Control Protocol (PCP)* [7] has been designed and recommended by IEFT to enable port forwarding in CGN deployments. However, our experiments have shown that most providers do not have this mechanism deployed at the moment and that some providers have deployed a version of CGN that does not satisfy NAT behavioral requirements [11], which makes P2P communication nearly impossible.

2.4. Freedom of Trustworthy Communication

The principle of network neutrality states that ISPs must treat all traffic equally and not discriminate based on its content, addresses of recipients, or methods of communication. However, we can see that this principle is violated in many cases, some more worrying than others. There are several countries enforcing Internet censorship either by simple IP address blocking, or advanced deep packet inspection, again preventing people from communicating freely.

Even in places with relatively mature infrastructure, there are occasionally connectivity issues at places with a high concentration of people, such as conferences or festivals, causing congestion in the infrastructure networks. The increasing capabilities of smartphones and novel wireless networking technologies open up possibilities to a whole new range of applications that can communicate without the need for the Internet connection. The ideal communication protocol should be able to use those when possible. It has been seen how the Bluetooth technology can be useful in creating a mesh network during the Hong Kong protests in 2018 and 2019, where protesters relied on peer-to-peer communication apps as a communication tactic.

Another problem related to communication is ensuring message authenticity and privacy. People want to be sure their communication is secure and that they are communicat-

ing with the person they think they are. Most of the commonly used protocols have adopted *Transport Layer Security (TLS)* protocol to secure all communication between clients and servers. The trust in the system is enforced by using *public key infrastructure (PKI)* with trusted certificate authorities. However, public key infrastructure is not easy to adapt to a peer-to-peer system with self-sovereign identities and it remains an unsolved problem.

2.5. Research Focus and Structure

While it is out of scope of this thesis to provide the complete solution for the next generation of web applications, the afore-mentioned problems and challenges lead us to the following question:

> **Can we create a technology stack that would allow building decentralized alternatives to big tech?**

The structure of this work is as follows. First, in Chapter 3, we explore the state of the art solutions for decentralized communication. In Chapters 4 and 5, we design a zero-server architecture and develop several networking primitives which remove the need for central control by any trusted third party. No central element exists which can form a performance bottleneck or point-of-failure. These serve as the basic building blocks for creating a full fledged Facebook alternative with full interoperability and data portability. Our Internet-deployed network primitives are:

- Identity layer based on public keys of trusted contacts with secure public key exchange
- Authenticated communication bypassing NAT boxes and other imperfect hardware
- Establishing connection with known contacts using a social-based overlay
- Secure end-to-end encrypted messaging of any message size
- Distributed bookkeeping with tamper-resilience to facilitate communities with self-governance using TrustChain

In Chapter 6, we then take advantage of these building blocks to build our decentralized version of a super app with the ability to extend its features using mini-apps. The first version of a distributed social network with private messaging and a public feed has been developed as the proof of principle that service like Facebook could potentially be built using this stack.

In Chapter 7 we evaluate various performance metrics of the implemented solution. Finally, in Chapter 8 we conclude the work and outline possible future work.

3

State of the Art

In this chapter, we conduct a short survey of various topics related to P2P communication. We start by analyzing existing solutions and platforms for decentralized communication. We continue by introducing the commonly used solutions for fixing the broken internet infrastructure by techniques known as NAT traversal. Finally, we introduce wireless communication technologies provided by modern smartphone devices and analyze them based on practical usability for infrastructureless communication.

3.1. Decentralized Communication Platforms

In general, we can divide communication platforms into three classes based on their architecture and degree of decentralization. The traditional *centralized* platforms such as Twitter, Facebook, and Instagram are operated by large profit-oriented companies. All data are stored in data silos owned by those companies and commonly used for targeting advertisements and other marketing purposes.

Decentralized platforms allow everyone to run their own node and be completely in control of their data. There are many projects dealing with the problem of P2P communication on different levels, ranging from general-purpose libraries that can be used to build applications (libp2p [16], IPv8 [35]), to feature-packed platforms providing social networking features on top of their custom-designed protocols (Briar [24], Secure Scuttlebutt [31]). We try to analyze their functionality from the technical and usability perspective, state their advantages and shortcomings, and propose improvements where possible.

Federated platforms represent the middle ground between centralized and decentralized platforms. They consist of an ensemble of interconnected servers that communicate with each other using open protocols.

3.1.1. Fediverse

Fediverse is an ensemble of interconnected servers that run open-source social networks and communicate with each other. Historically, there have been many proposals of open protocols for social networking, and many projects built on top of them. In 2018, the World Wide Web Consorcium presented ActivityPub [37], a protocol aming to improve interoperability between platforms. Some of the most known platforms implementing this protocol are Friendica[1], PeerTube[2], or Mastodon[3]. The screenshot of the latter is shown in Figure 3.1.

[1] https://friendi.ca/
[2] https://joinpeertube.org/
[3] https://mastodon.social/

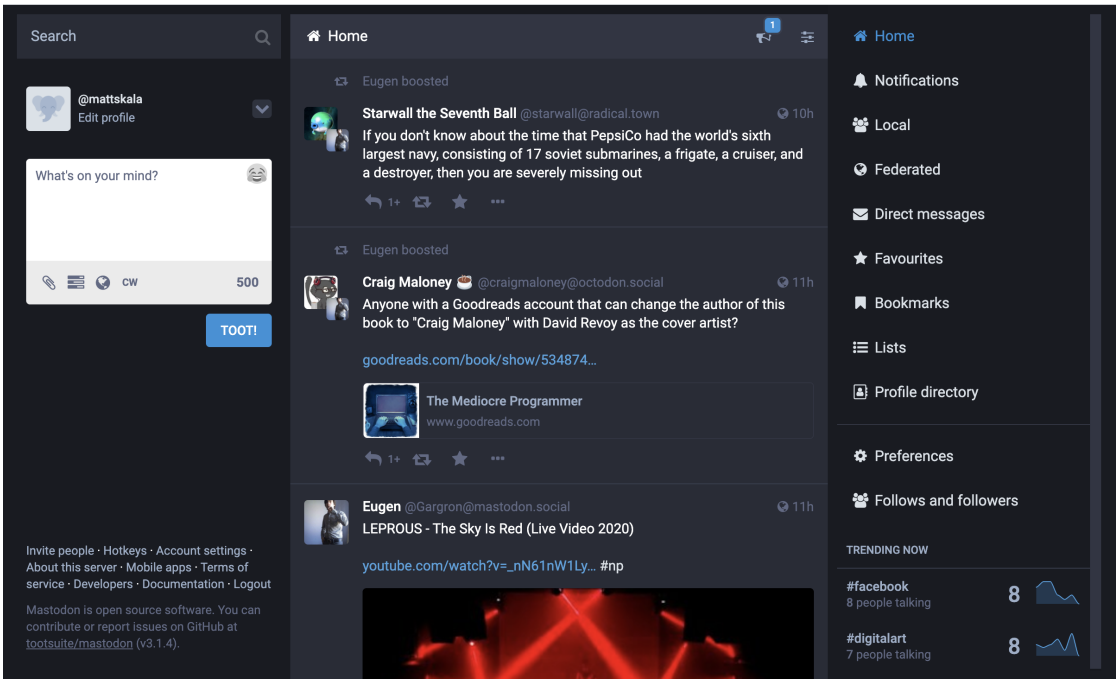

Figure 3.1: The screenshot of mastodon.social, an example of a federated social network

In contrast to mainstream social platforms, federated networks are developed by communities of people, independent from any corporation. Anyone can run its own Fediverse server, and users can choose the operator through which they are connecting to the network. However, users still have to a certain extent trust their server operators. There is also usually a lack of incentivization for server operators, and some operators charge a fee for joining the network to fund their operation.

3.1.2. Briar

Briar [24] is an open-source project which aims to support freedom of expression and right to privacy. It enables peer-to-peer encrypted messaging and forums. It is presented as a tool for activists, journalists, and anyone who needs a safe way to communicate.

Before the communication starts, users have to meet in person and scan QR codes from each other's screen. The devices exchange public keys and agree on a shared key using *Bramble QR Code Protocol (BQP)*. This provides strong identities secure against man-in-the-middle attacks. The device then only accepts connections from devices in contacts. The user can also initiate an introduction between two of her contacts. If both contacts accept the introduction request, then they are able to establish connections without meeting in person.

The communication is built on top *Bramble Transport Protocol (BTP)* and *Bramble Synchronization Protocol (BSP)*, transport and application layer protocols suitable for *delay tolerant networks* [2]. It does not rely on any central server, but instead allows to synchronize messages using Bluetooth or Wi-Fi. If the Internet is available, it can also connect via the Tor network. Screenshots of the Briar app UI are provided in Figure 3.2.

3.1.3. Secure Scuttlebutt

Scure Scuttlebutt (SSB) [31] is a peer-to-peer gossip protocol for building decentralized applications. This is the work that seems to be the most closely related to our research and shares many similar ideas, including a distributed append-only data structure. The first application built on top of the protocol has been a distributed social networking platform Scuttlebutt.

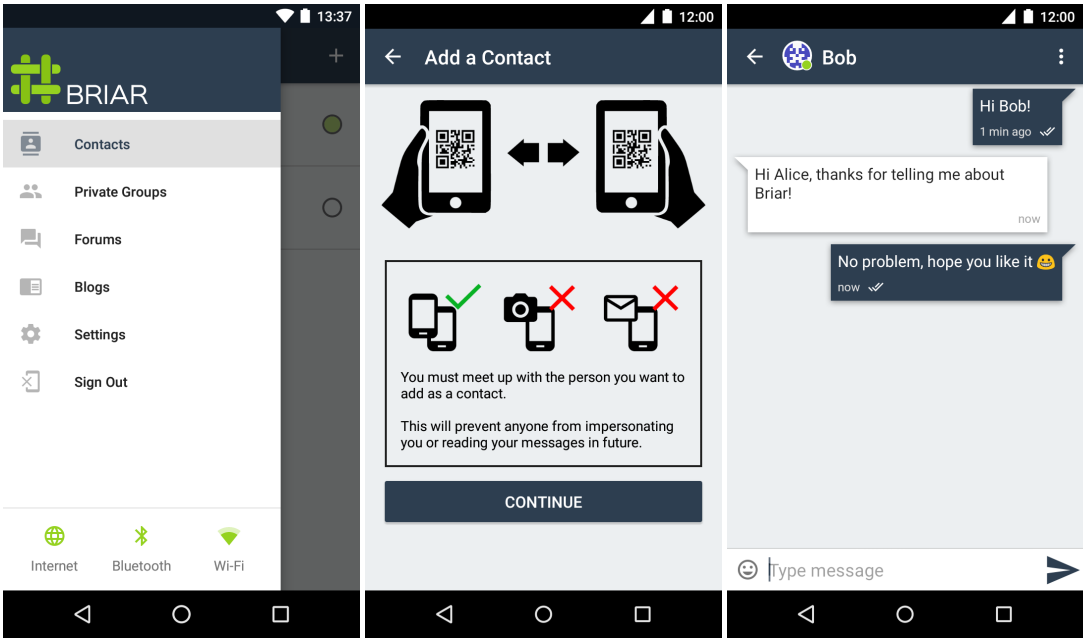

Figure 3.2: The menu, contact exchange, and the conversation detail in the Briar app

Each peer in the network has its own identity generated from its public key. Every identity is tied to its own feed, which is represented by an append-only log of messages. Each message is signed with the peer's private key to provide authenticity. Every message also has a pointer to the hash of the previous item in the log, which helps to ensure data integrity during replication. The system architecture is designed to be completely decentralized and work off-grid without any infrastructure. Each peer stores its entire feed history and feeds received from other peers. When peers get connected, they exchange their feeds and feeds of people they follow. The protocol provides eventual consistency.

The protocol can perform peer discovery either by UDP broadcasting over the local area network, or more commonly, by using *pubs*. A pub is a peer which is publicly accessible on the Internet. The user who wants to join the network first needs to obtain an *invite link* containing the IP addres, port, and a public key of a pub. The peer establishes a connection with the pub over TCP using the Secret Handshake key exchange [30]. The peer can then communicate with the pub and access feeds of other peers followed by the same pub.

Messages can contain links to binary large objects, or *blobs*. Peers send to each other lists of blobs they *have* and *want*, in a similar fashion as in the BitTorrent protocol. Peers can also download blobs on behalf of each other if any of their peers have the blob requested by another peer, to increase data availability. The blobs are addressed by hashes, so anyone who fetches them can verify they have not been tampered with.

Sometines the user wants to send a private message that can be read only by one or more specific people. In that case, the message needs to be encrypted in a way that it can be decrypted only with people owning the corresponding private keys. For each private message, the sender generates a random secret key to encrypt its content. The sender then encrypts the secret key with a public key of each recipient of the message and includes these encrypted keys in the message header. The identity of recipients is not revealed. Instead, every peer who receives the message has to try to decrypt the key to find out if they are among the intended recipients.

While the protocol claims to be peer-to-peer, it is based on client-server communication model, as most of the communication happens via publicly available peers called pubs.

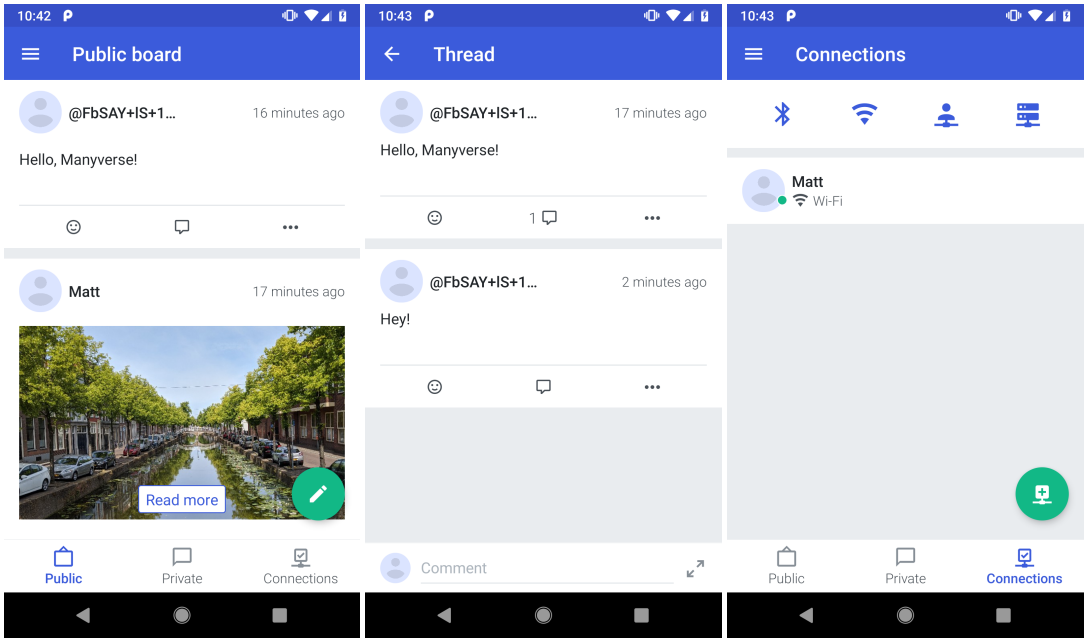

Figure 3.3: The feed, thread, and connections list UI in the Manyverse Android app

There is no NAT traversal layer, so peers behind NATs are not able to communicate with each other directly. We are also questioning the scalability and incentive alignments of the protocol, as users are by default supposed to store complete feeds and blobs even for friends-of-friends. Also, all private messages are stored in the public feed and replicated across many peers, even when the message can only be decrypted by a single peer, which creates unnecessary overhead.

3.1.4. libp2p

libp2p [16] is a modularized peer-to-peer networking stack and library. It is being developed by Protocol Labs, a non-profit company behind the Interplanetary File System (IPFS)[4], a distributed filesystem combining some fundamental ideas from BitTorrent and Git. They extracted the networking logic from the IPFS project into a library after realizing it could be useful for other applications as well. It is to date probably the most widely known universal P2P library. It was originally implemented in Go and JavaScript, and there are ongoing efforts to provide implementations in Rust, Haskell, Kotlin, and Python. One of the promises of the Kotlin implementation will be the ability to run it natively in Android runtime.

The library is composed of several layers, where each can be implemented by one of interchangeable modules. It is presented as a collection of peer-to-peer protocols for finding peers, connecting to them for finding content, and transferring it. It implements port mapping and hole punching inspired by STUN to deal with NATs. The *identify* protocol allows nodes to discover their public address and the *AutoNAT* service allows them to discover the NAT behavior by forcing other peers in the network to attempt to connect to them. When the NAT traversal fails, there is a circuit relay protocol that allows peers to communicate indirectly using intermediate peers, which similar to the functionality of TURN servers. Currently, the relay protocol requires a list of relays to connect to. An *autorelay* protocol for discovering public relays is under active development. *Kademlia DHT* is used as a primary mechanism for peer routing, i.e. finding routable addresses of peers based on their public keys.

[4]https://ipfs.io

3.1.5. IPv8

IPv8 [35] is a P2P networking library developed over the last 13 years at TU Delft. It is an evolution of previous generations of *BuddyCast* [23] and *Dispersy* [40]. Its primary purpose was to serve as the networking layer for Tribler [22], an anonymous P2P file-sharing client. However, it attempts to be a general-purpose library which can be used to build custom P2P overlays and applications. It is conceptually composed of several layers.

The identity layer provides each peer with a private and public key pair. This allows authenticated communication. The public keys can also be used for addressing and IP addresses are ultimately abstracted away. UDP hole punching is implemented as a basic NAT traversal technique to allow connectivity between peers behind NATs.

Discovery protocol based on a *distributed hash table (DHT)* allows to connect to specific peers using their public key, without need for knowing their IP address. While there is a list of trusted bootstrap servers provided, in theory any peer in the network can be used to bootstrap connection, thanks to distributed nature of UDP hole punching protocol which does not rely on STUN servers.

The library also provides the implementation of TrustChain [21], a scalable distributed ledger for tamper-proof accounting. It is currently mainly used as a bandwidth accounting mechanism to prevent freeriding in Tribler, but has wide range of other potential use cases, including a decentralized asset exchange [8], or identity attestations and verifiable claims.

It is written in Python, as well as the rest of the Tribler stack. While using a single language during the whole development process is convenient for the Tribler team, the language choice has been a limiting factor when porting to mobile platforms such as Android. There have been attempts to run IPv8 on Android using *python-for-android*[5] toolkit which allows to build a Python interpreter together with the application source code into an APK. However, the whole build process has shown to be fragile and impractical. Firstly, the build of the whole library takes almost an hour, which severely degrades developer experience and slows down iterative development. Secondly, it has turned out to be problematic to compile a 64-bit APK that would comply with the requirements for publishing on Google Play.

There has been a long track of efforts to explore the usage of mobile devices in peer to peer systems at TU Delft. The **app-to-app communicator**[6] was an early prototype that implemented a basic UDP hole punching method to test the feasibility of a device to device overlay without any server infrastructure. The **self-compiling Android application**[7] was an experimental project that bundled all tools from the Android SDK required to assemble an APK on Android. This has proven viability of trustworthy code execution without dependency on a centralized app store. The **trustchain-android**[8] project implements a TrustChain-like distributed ledger and a UDP hole punching protocol. It has been written in Java, but the protocol is not compatible with TrustChain which was originally implemented as part of the py-ipv8 library, even though it is based on the same principles.

3.2. NAT Traversal

Network Address Translation (NAT) is a method of mapping addresses from one address space into another by modifying the IP header of a packet on its way from the source to the destination. It has been originally proposed as a temporary solution to slow down IPv4 address space exhaustion, until a better solution is implemented. [10] Yet, it has become so ubiquitous in the Internet infrastructure that almost all end users are located behind a NAT nowadays.

[5] https://github.com/kivy/python-for-android
[6] https://github.com/Tribler/app-to-app-communicator
[7] https://github.com/Tribler/self-compile-Android
[8] https://github.com/Tribler/trustchain-android

NAT is designed to be fully transparent when using the client–server communication model. However, it creates major obstacles to peer-to-peer and VoIP applications, where end user devices need to establish direct connections without using a third-party server. This topic has been extensively researched and still, fragile workarounds are required to support direct communication between users behind NAT boxes. This hurdle could be one of the reasons why the server-centric model, which deviates from the original ideology of the Internet, has prevailed in most of the services we use today.

NAT traversal refers to a set of techniques used to establish a connection between devices behind NATs. Usually, a help of a third party is needed to establish a connection, but the subsequent communication takes place directly between interested devices. Most of the solutions are based on the UDP transport protocol, due to its simplicity. While there are NAT traversal methods for TCP as well, they are much more complex and less reliable, so we will not consider them further.

3.2.1. Background on NAT Classification

The original NAT proposal did not specify the exact behavioral requirements of a NAT, so hardware manufacturers had to decide a lot of implementation details on their own. This has resulted in fragmentation and many different NAT behaviors with different levels of restrictions. However, in general, most NATs can be classified according to their *address and port mapping* and *filtering* behavior [11].

We define an *endpoint* as a pair of an IP address and port. When a packet is sent by an internal endpoint located behind a NAT to any endpoint on the public Internet, the internal endpoint is mapped to an external endpoint and a mapping is created on the NAT box. We can then classify the following mapping behaviors based on how mappings are reused when communicating with different endpoints:

- **Endpoint-Independent Mapping (EIM)**: The NAT reuses the mapping for any packets sent from the same internal endpoint to any external IP address and port.
- **Address-Dependent Mapping (ADM)**: The NAT reuses the mapping for packets sent from the same internal endpoint to the same external address (using any external port).
- **Address and Port-Dependent Mapping (APDM)**: The NAT reuses the mapping only for packets sent from the same internal IP address and port to the same external address and port.

When an endpoint sends a packet to our external endpoint, it gets translated based on the mapping stored by the NAT. However, NAT can also discard any incoming packet based on its filtering behavior:

- **Endpoint-Independent Filtering (EIF)**: The NAT filters out only packets not destined to the internal endpoint. Therefore, when an internal endpoint creates a mapping by sending a packet to any endpoint, it can then receive packets from any endpoint sending a packet to its external address and port.
- **Address-Dependent Filtering (ADF)**: The NAT only allows incoming packets from an address to which an internal endpoint has previously sent a packet.
- **Address and Port-Dependent Filtering (APDF)**: The NAT only allows incoming packets from an address and port to which an internal endpoint has previously sent a packet.

While EIM is the least restrictive mapping behavior, it is also the required behavior by [11] to ensure the correct functionality of many applications. It has been previously measured that around 79% peers on the Internet are not directly connectable and require further mechanisms to establish connectivity. Also, 11% of peers were found behind a NAT with AP(D)M which does not support common NAT traversal mechanisms [12]. However, it can be hoped that the manufacturers start to comply with the NAT behavioral requirements, so the number of non-connectable nodes will decrease over time as NAT boxes are upgraded and networks transfer to IPv6.

3.2.2. Hairpinning

Hairpinning is a mechanism that allows two hosts located behind the same NAT to communicate with each other using external addresses. The NAT should correctly recognize such packets and re-route them back to the local network. While it is required by [11] for NATs to implement this behavior, its support varies among manufacturers are NAT box models. Therefore, a robust networking library should not rely on this property and should use LAN addresses to communicate with peers on the same local area network.

3.2.3. Carrier Grade NAT

Traditionally, each consumer had a NAT implemented in the router located at the edge of the network. However, to further conserve the address space, and facilitate transitions to IPv6 with backwards compatibility of IPv4, it has become a trend among internet service providers to implement a NAT in their infrastructure. This topology is called a *carrier grade NAT (CGN)*. It is especially used in mobile networks, as it allows all subscribers connected to the same gateway to share a pool of private network addresses which are then translated by NAT to an external address.

When the CGN is deployed in home broadband, there usually already is a NAT implemented on consumer premises. This scenario is then called *NAT444*, as the address gets translated twice along the route and the packets pass at least through 3 different addressing domains: the customer's private network, the carrier's private network, and the public Internet. Another common topology is *Dual-Stack Lite (DS-Lite)* which can be used as a transition mechanism from IPv4 to IPv6. In DS-Lite, the carrier's network uses IPv6 addresses which get translated to IPv4 for the public Internet.

CGN is by definition managed by the network operator, and the customer does not have any control over it. Its implementation is usually complex enough to provide scalability and ensure reliability for thousands subscribers and compliance with legal regulations, which again violates the end-to-end principle by keeping too much state in the network. Inability to manage open ports breaks the communication model of many peer-to-peer applications which usually have to reside to using a proxy server for relaying communication.

3.2.4. Port Forwarding

Seemingly the most correct way to allow incoming traffic would be to manually create a mapping in the NAT and allow incoming traffic for the mapped port in the firewall. However, this requires considerable effort from the user side and it cannot be expected that regular users are also network administrators.

Several protocols have been proposed for automatic port forwarding configuration which could be used by applications to open desired ports without user interaction and find out the mapped public address.

One of such protocols is *Universal Plug and Play (UPnP) Internet Gateway Device Protocol (IGDP)*. It allows to list existing port mappings, add or remove them, and learn an external

IP address. It is sometimes used by small home or office networks. However, the protocol is not authenticated, and one computer could request a port mapping for another one. It carries some security risks which have been exploited in the past, including the infamous *Flash UPnP Attack*. Another group of issues is caused by improper protocol implementation in routers, which has e.g. allowed to re-route all traffic from the internal network to an external server. [13]

NAT Port Mapping Protocol (NAT-PMP) is a similar protocol introduced by Apple which has been implemented by various Apple products and it was still primarily focused only on home gateways. NAT-PCP was superseded by the *Port Control Protocol (PCP)*, which was standardized in 2013. [7] It allows for deployment in various scenarios, including carrier grade networks.

Using port forwarding sounds like a promising approach that would allow applications to control open ports and learn public addresses reliably without using any third parties. However, in practice, these protocols are commonly disabled, possibly for security reasons. We could argue that modern port forwarding protocols such as PCP do not reduce security in any way, as the same effect can be achieved with other NAT traversal methods, though much more expensively.

3.2.5. Session Traversal Utilities for NAT (STUN)

STUN was originally defined in [25] as *Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. It was a client–server protocol that allowed applications to discover the presence and types of NAT, and determine the public IP assigned to them by the NAT. However, it has been shown that the protocol does not work reliably enough to be a deployable solution. The NAT classification used by the STUN did not cover all possible types of NATs, and it did not provide any remedy in scenarios where the peers were not connectable.

The protocol has been made obsolete by the introduction of *Session Traversal Utilities for NAT* in [26]. It no longer presents itself as a complete solution to NAT traversal, but rather as a tool that can be used by other protocols such as ICE to discover the public IP address. This process is also known as *Unilateral Self-Address Fixing (UNSAF)*. It is a completely new protocol with the same name, which causes some confusion, and when dealing with different implementations, one has to make sure which version of STUN it actually implements.

3.2.6. Traversal Using Relays Around NAT (TURN)

While STUN can be used to discover the public addresses that peers can use to communicate, it does not guarantee these are addresses can actually be used for communication. Specifically, a symmetric NAT is known not to be compatible with STUN. *Traversal Using Relays Around NAT (TURN)* provides a solution that is guaranteed to work with any NAT. It is based on a TURN server which is used to relay the traffic between two peers.

The protocol allows a host behind a NAT *(TURN client)* to request another host *(TURN server)* to act as a relay. Upon request, the server allocates an address which can then be advertised by the client and used to communicate with multiple peers. When a peer sends a packet to the server address, it is relayed to the appropriate client. When a client sends a response to the server, it is sent to the peer on behalf of the server.

While this approach allows to establish connection in almost all scenarios, it comes with a high cost on the TURN server operator, so it is meant to be used only as a last resort when no other type can be established. There is also no incentive for TURN server operators to provide this service.

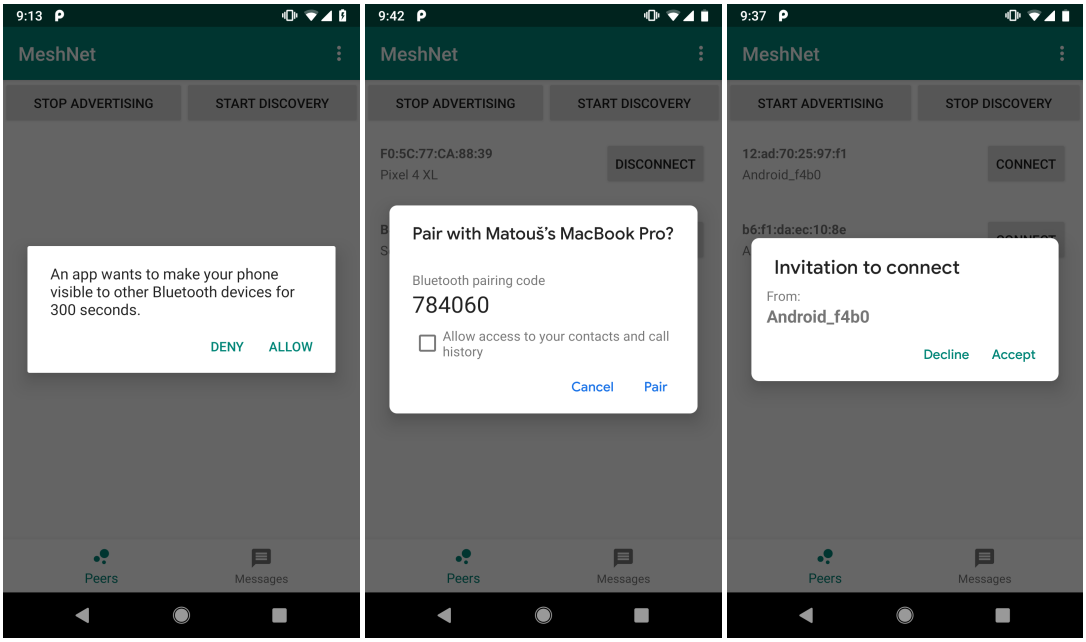

Figure 3.4: The system UI for enabling Bluetooth discovery, accepting an incoming Bluetooth pairing request, and accepting an incoming Wi-Fi Direct connection on Android 9

3.3. Infrastructure-Less Communication

Modern smartphone devices come equipped with several wireless communication standards that can potentially be used for communication with other nearby devices. It is desirable to use such a technology when multiple devices in proximity want to communicate with each other when there is no reliable Internet infrastructure available. These technologies can be also preferred over the Internet in case of censorship and privacy concerns, as has been shown during numerious occasions such as Hong Kong protests. From the user experience perspective, it is desired that the device discovery and connection establishment does not require any user interaction besides the one required by the application use case. However, this is often difficult to achieve in the security model of smartphone operating systems, which try to protect users by enforcing a system UI for any sensitive operations, as shown in Figure 3.4.

3.3.1. Bluetooth

The oldest and the most battle-tested technology for nearby connectivity is Bluetooth, which has been in development for more then 20 years. The common flow for Bluetooth usage is to first force the user to *pair* (or *bond*) two devices. The device A first needs to manually be set as *discoverable*, usually for a limited time period. The device B then performs a scan to discover nearby devices. The device B can then send a pairing request to the selected device. Then a pairing code is displayed and once both users accept the pairing request, the devices are paired.

Only after that, a secure *Radio frequency communication (RFCOMM)* channel can be established. The pairing process requires user interaction, which degrades the user experience in certain applications when authentication is performed on the application level. While it is possible to establish an *insecure* RFCOMM channel if the MAC address of the other device is known, the user of the other device still needs to manually set it to be discoverable.

Technology	Android	iOS	Throughput	Range
Bluetooth	2.0+	5.0+	2 Mbps	~40 m
BLE Advertising	4.3+	6.0+	0.3 Mbps	~100 m
BLE GATT	5.0+	6.0+		
BLE L2CAP	10.0+	11.0+		
Wi-Fi Direct	4.0+	N/A	250 Mbps	~200 m
Wi-Fi Aware	8.0+	N/A		

Table 3.1: The comparison of properties of the most common wireless communication technologies and their support in smarphone operating systems

3.3.2. Bluetooth Low Energy

Bluetooth Low Energy (BLE) [34] was introduced in 2010 as part of the Bluetooth 4.0 specification. It is a completely different communication protocol incompatible with the classic Bluetooth. BLE offers considerably decreased power consumption with a similar communication range and slightly lower bandwidth. It was originally intended to support an infrequent low-power communication with wearables, healthcare accessories, or smart home appliances. However, while it is not a primary use case, it could also be potentially used for low-bandwidth peer-to-peer communication between smartphone devices.

It is notable that once an application is granted a Bluetooth permission, it can fully control BLE APIs without any user interaction, which opens up doors for a range of many different applications.

3.3.3. Wi-Fi Direct

There have been many attempts to enable direct communication between IEEE 802.11 radio devices. The 802.11 standard defines two operating modes. Next to the traditional *infrastructure* mode, there is an *ad-hoc* mode which allows device-to-device communication. However, the ad-hoc mode is not supported by Android OS, even though it can be enabled on some devices with a root access.

Wi-Fi Direct (also known as Wi-Fi Peer-to-Peer) [38] is a IEEE 802.11 based protocol released by Wi-FI Alliance in 2009 and supported from Android 4.0. With Wi-Fi direct, devices are organized in groups, where one device is the Group Owner (GO) and the rest are Group Members (GM). The roles are not predefined, but are negotiated during the group formation process. Groups are able to support Legacy Clients (LC), which means that even devices without Wi-Fi Direct support can join as group members.

3.3.4. Wi-Fi Aware

Wi-Fi Aware, also known as *Neighbor Awareness Networking* (NAN) is a recent networking standard introduced by Wi-Fi Alliance. [39] It works by forming clusters with nearby devices. The discovery process starts when one device (a *publisher*) publishes a discoverable service. Other devices (*subscribers*) who subscribe to the same service will receive a notification once a matching publisher is discovered. After the subscriber discovers a publisher, it can either send a short message or establish a network connection with the device. A device can be both a subscriber and a publisher simultaneously.

4

Protocol Design

Any alternative to Big Tech requires a permissionless communication protocol. In this chapter we explain our design. We start by describing common problems in P2P networks such as identity, peer discovery, data transport, and NAT traversal. For each of the problems, we discuss the solution used by our protocol.

It is important to note that in this thesis, we extend an existing IPv8 protocol [35]. It is desirable that our protocol is backwards compatible with the original py-ipv8 implementation, so that we can connect and communicate with the existing peers in the network and use the services implemented by the existing infrastructure. For that reason, a lot of design decisions that have been previously made had to be accepted as such, even in cases where a different approach would probably be taken in case we designed the protocol from scratch.

4.1. Identity and Keys

In the Internet Protocol, each computer gets assigned an address which is subsequently used for packet routing. The most common version of the protocol, IPv4, uses a 32-byte address space which is not enough to uniquely identify all devices on the planet. To deal with address exhaustion, internet providers were forced to deploy *Network Address Translation (NAT)* which allows a single address to be shared across multiple devices. With the rise of portable computers and smartphones, using IPv4 addresses on the application level is problematic as they are dependent on the physical location and should not be considered stable user identifiers. *Mobile IP* [3] is one of the proposals that allows devices to move between networks while maintaining their IP addresses, but its wide adoption is unlikely due to the architectural complexity.

Our goal is therefore to define custom peer identities on the protocol level and abstract IP addresses away from the applications. We also want to be able to sign messages to provide authenticity, and optionally encrypt them so they are readable only by the intended recipients. Both issues can be solved by public key cryptography. We will use elliptic-curve cryptography based on the on *Curve25519*, which has been a de facto industry standard for past few years due to its security properties and absence of restricting patents.

Prior to joining the network, each peer needs to generate a *private key* represented by 32 bytes. It can be used for signing messages to prove their authenticity, or for decrypting encrypted messages. The private key can be in turn used to derive a *public key* by multiplying the generator point of the curve by the private key. This operation is known to be irreversible, so it is computationally infeasible to obtain the private key from the public key. The public key can be shared with other peers and is used to verify that the message has been indeed

created by the one who claims to be its sender. Furthermore, it can also be used for encrypting messages that will only be readable by the recipient. Finally, we calculate the hash of the public key to derive the *peer ID*. The peer ID can be used to visualize the identity in the application UI as it is shorter than the public key, but the protocol usually operates with public keys directly. This simple process is shown in Figure 4.1.

Figure 4.1: Public key generation process

4.2. Peer Discovery

Peer discovery is the process of finding other peers in the network we should connect to. It is an essential mechanism in P2P systems as it allows new nodes to join the network.

4.2.1. Bootstrap Server

Probably the most common approach is using one or more bootstrapping servers that act as trackers. The bootstrap server maintains a list of peers in the network and maintains the list of online peers with regular keep-alive checks. This is the only central component in our system. Ideally, it should be used only for finding the first few nodes, and after that, the peer discovery would continue using these nodes, without contacting the boostrap server again.

4.2.2. Multicast DNS

When multiple peers are connected to the same local area network and they want to communicate with each other, there should be no need to contact an external server. We could utilize UDP multicast packets to notify other peers in the network who are in the same multicast group. This is the basic principle of the *Local Peer Discovery* protocol used by BitTorrent. Another option would be to use *multicast DNS* (mDNS) mechanism in conjunction with *DNS Service Discovery* (DNS-SD), which is a zero-configuration protocol using packet formats of a DNS system. It allows us to discover local peers providing the given service and connect to them. We use mDNS, as it is natively supported by Android SDK.

4.2.3. Bluetooth Advertising

When peers are in proximity but not connected to the same network, they can still discover each other with some nearby connectivity technologies, such as Wi-Fi Direct, Wi-Fi Aware, or Bluetooth. We have selected Bluetooth Low Energy as it is the most energy efficient and most widely supported technology in today's smartphones. A peer that has Bluetooth turned on is actively broadcasting their identity over advertising packets that can be discovered by other devices. The complete mechanism for nearby communication will be described more in detail in later sections.

4.3. Transport Layer

The transport layer is responsible for transferring data between peers. There are two transport layer protocols implemented in the IP suite: *Transmission Control Protocol (TCP)*, and *User Datagram Protocol (UDP)*.

4.3.1. Reliable vs. Unreliable Transport

TCP provides *connection-oriented*, *reliable* streams. That means that the connection needs to be established before any communication happens. This is done by means of a two-way handshake. Once the connection is established, messages can be exchanged. The messages are acknowledged, so the communication is reliable. The sliding window is used for congestion control, so multiple messages can be on the fly at the same time, ensuring efficient usage of the available bandwidth.

On the other hand, UDP is a simple *connectionless* protocol without any delivery guarantees. UDP packets are called *datagrams* and their behavior can be described as *fire-and-forget*, as the sender generally does not have any information about packet delivery. It is primarly intended for use in real-time applications where packet loss is acceptable.

Many commonly used protocols such as HTTP take advantage of the reliable transport provided by TCP. However, in P2P setting, UDP is usually favored thanks to its simplicity and absence of a handshake, which makes NAT traveral easier. Interactive Connectivity Establishment (ICE), a common NAT traversal method, has been originally defined only for UDP. Though extensions for TCP have been proposed, they rely on a specific NAT behavior to work and have significantly lower success rate than over UDP. For these reasons, we will also choose to use UDP.

4.3.2. UDP Socket Multiplexing

Multiplexing is a method of combining multiple communication channels to share a single communication medium. In P2P communication, every peer commonly communicates with many peers simultaneously and thus needs to keep multiple open connections. The TCP/IP stack implements multiplexing by using different ports for different sockets. However, we implement our own multiplexing mechanism that allows us to use a single UDP socket for all communication. Not only it removes the overhead of keeping multiple open sockets at the same time, but more importantly, we have to deal with the NAT traversal only once, and our publicly facing address stays constant for all peers.

4.3.3. Secure Communication

We consider the communication between two parties to be secure when both parties can be sure no one is altering or intercepting the communication. This can be ensured by the means of cryptographic techniques authentication and encryption. By default, all packets are signed to provide authentication. However, as every application has different needs, this behavior can be easily overriden by the application developer to use encryption or authenticated encryption when desired.

4.3.4. Binary File Transfer over UDP

The size of a UDP packet is theoretically limited to 65,535 bytes. However, the size is in practice limited by the *Maximum Transmission Unit (MTU)* of Ethernet, which is 1500 bytes, minus 60 bytes taken by the IP header. With 8 more bytes taken by the UDP header, this leaves only 1432 bytes available for the UDP packet payload. When the user tries to send a larger packet, it is fragmented and delivered in parts without any ordering or delivery guarantees. Therefore, when working with UDP, the application developer needs to make sure to stay within this limit.

This is not practical, and a general-purpose networking library should abstract developers away from the underlying communication protocol. Ideally, we should be able to send messages of unbounded size. There are many binary transfer protocols that can be implemented on top of UDP, including *TCP over UDP*, *uTP*, or *QUIC*.

There is no binary file transfer protocol implemented in `py-ipv8`. In our implementation, we decided to use the *Trivial File Transfer Protocol (TFTP)*, due to its simplicity and ease of implementation. It is a simple protocol that splits the payload into the chunks of 512 bytes and then keeps sending them one by one. It awaits for an acknowledgement of each packet before sending the next one. This makes it very inefficient with high bandwidth and considerable network delays, as can be seen in the later evaluation chapter. While it was suitable for a prototype and improving the reliability of the transport layer, in the future, a more efficient protocol with a sliding window should be implemented instead.

4.4. NAT Traversal with Peer Introductions

One of the traditional NAT traversal methods called *UDP hole punching* allows to establish UDP connectivity between two peers when both of them are hidden behind a NAT. It is based on the concept that both peers fire a UDP packet targeted at each other at the same time. This first packet creates a mapping entry in the sender's NAT and allows subsequent incoming packets to be delivered. This process can also be seen as opening a *hole* in the firewall, which explains the term *hole punching*.

IPv8 implements a decentralized variant of UDP hole punching mechanism integrated with the peer discovery process, which has been previously described in [12] and [40]. The complete peer discovery protocol consists of 4 messages and it is visualized in Figure 4.2.

After reviewing the py-ipv8 implementation, it has been discovered that it only works reliably when all nodes are behind different NATs. This is because of the fact that the tracker only stores WAN addresses which are then used for peer introductions, and peers always use their WAN address for communication. When multiple nodes are located behind the same NAT, they can still commnicate in case their NAT supports hairpinning, but this is not always the case. There is no mechanism to discover other peers on the same LAN without using a tracker, and there is no way to use LAN addresses for communication. We will fix this flaw by reusing some additional fields which are already present in the IPv8 packet format probably for legacy reasons, but are not currently used. Thanks to that, the updated protocol is still backwards compatible and can be used to connect to peers using older versions of the protocol. The updated protocol works as follows:

1. When Alice wants to connect to a new peer in a specific community, it can send an *introduction request* to a tracker, or any other peer present in the community. The introduction request should contain the community ID representing the ID of the community in which Alice wants to find a new peer, and her LAN and WAN address.

2. We assume Alice sends an introduciton request to Bob. Bob selects a random peer (Charlie) from the requested community, and sends a *puncture request* to him. The puncture request contains the WAN address of Alice.

3. At the same time, Bob sends an *introduction response* back to Alice. The introducion response contains LAN and WAN address of Charlie.

4. As soon as Charlie receives a puncture request, he should send out a packet to the WAN address of Alice. That will create a mapping in his NAT, so Alice would be able to contact him.

5. Alice then keeps sending introduction requests to Charlie's WAN address for a specified interval. After Charlie sends out a puncture packet, the next introduction request should reach him. She can also try to contact him over LAN if they reside in the same subnet.

6. Charlie should respond to the introduction request with an introduction response, in which he include another peer for introduction. Upon receiving an introduction request or response, the receiver should add the sender to their verified peer list.

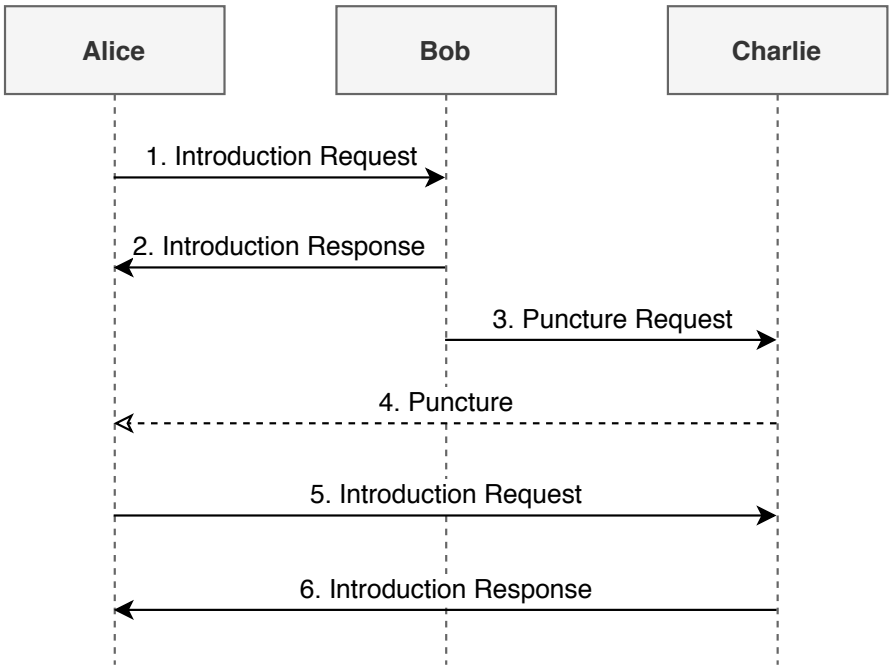

Figure 4.2: Peer discovery mechanism with NAT traversal

4.5. Symmetric NAT Traversal

Our method provided in the previous section allows us to traverse around NATs that perform endpoint-independent mapping. However, as previously discussed in Section 3.2.1, peers behind NAT with address-dependent mapping would only be able to connect to public peers that do not require NAT traversal. This poses a major threat to our goal of global connectivity.

4.5.1. Topological Assumptions

We have extended our peer introduction protocol to work with some additional NAT behaviors that we encountered during our experimental evaluation of CGN deployments. For further discussions, we consider three different cases with respect to NAT topology between two different peers A and B:

Case 1: One peer behind a symmetric NAT

1. Peer A is behind a NAT with endpoint-independent mapping.

2. Peer B is behind a NAT with endpoint-independent IP address mapping, but an arbitrary port mapping behavior.

3. Peer B can estimate a session limit implemented by their NAT. By default, the algorithm first assumes there is no limit. In case the first traversal attempt fails, then it falls back to the default limit of 1.000 sessions with a 30-second timeout. This limit has been empirically shown to be successful in all scenarios considered in our experiments. However, the peer should have an option to tweak this parameter if they believe their NAT implements more permissive or restricted behavior.

Case 2: Both peers behind symmetric NATs without a session limit

1. Both peers are behind a NAT with endpoint-independent IP address mapping, but an arbitrary port mapping behavior.
2. There is no session limit implemented by any of the NATs.

Case 3: Both peers behind symmetric NATs with a session limit

1. Both peers are behind a NAT with endpoint-independent IP address mapping, but an arbitrary port mapping behavior.
2. There is a session limit implemented by one or both of the NATs.

4.5.2. NAT Type Detection

We would like to have a fully automated NAT traversal. For that, we need a mechanism to decide whether to perform a simple UDP hole punching, or there is a need for an extended multiple UDP hole punching method. We try to detect the NAT type based on its behavioral properties. We keep a log of triples consisting of our LAN address, our WAN address, and peer address that reported our WAN address. When we detect our LAN address has been mapped to multiple WAN addresses as reported by different remote peers, we can conclude we are located behind a symmetric NAT. Otherwise, we report our NAT type as *unknown*, as it is not necessary to know the exact NAT type for the purposes of UDP hole punching.

4.5.3. Extended Peer Introduction Protocol

The flow diagram of the extended peer introduction protocol is visualized in Figure 4.3. In our scenario, we consider Case 2, but it is trivially adjustable to Case 1 by limiting puncture rate and performing multihole punching only in one way. The protocol works as follows:

1. Alice sends an introduction request to Bob, and indicate she is behind a symmetric NAT.
2. Bob sends a puncture request to Charlie, and indicate Alice is behind a symmetric NAT.
3. Charlie starts sending puncture packets to Alice. As Charlie does not know the mapped port of Alice, it sends packets to all ports in the range of 1024 to 65535.
4. At the same time, Charlie sends a puncture response back to Bob.
5. Bob receives the puncture response from Charlie and sends the introduction response to Alice.
6. Alice receives the introduction response indicating that Charlie is behind a symmetric NAT. It starts sending puncture packets to Charlie, again trying all possible ports in the valid range. Eventually, at least a single puncture message should come through.
7. Whenever Alice or Charlie receives a puncture message, it should send an introduction request back.
8. The other peer responds with an introduction response in a regular way.

Our method will work in Case 1 and 2. In Case 3, we would need to send packets at the frequency restricted by the session limit. However, as port mappings can frequently change on both sides due to a binding timeout, there is no guarantee when both peers will find find the port mapping. We can keep trying opening ports at random, hoping both peers will meet

eventually. It is questionable whether it makes sense to support this case, as the effort of establishing the connection might not meet the benefits. It would make sense for long-lived connections that are kept alive for a long period. However, we expect mobile connections to be mostly temporary, so this case has not been implemented while it is theoretically feasible.

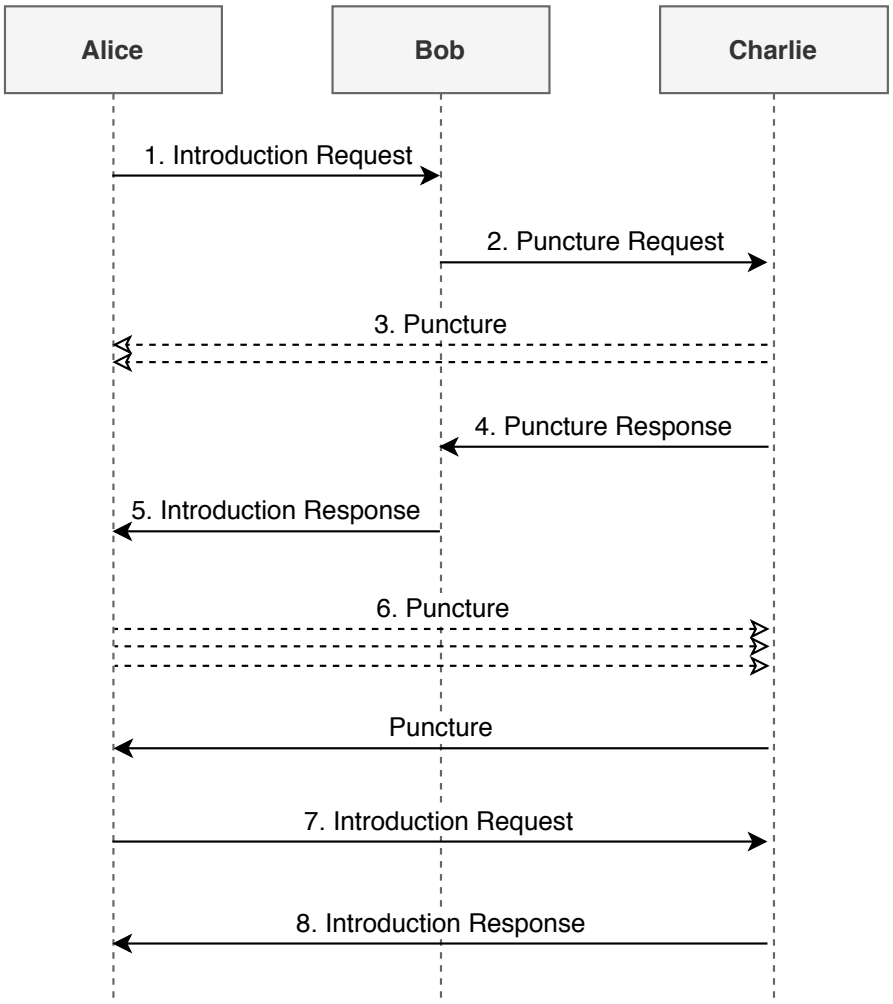

Figure 4.3: The flow diagram of the symmetric NAT traversal protocol

4.6. Communication with Nearby Devices

In Section 3.3 we analyzed different methods of infrastructureless communication on smartphone devices. For our purpose, it has been found that Bluetooth Low Energy is the most suitable technology thanks to the range of supported Android versions, and its ability to connect to new devices without any pairing and user interaction.

4.6.1. Introduction to Bluetooth Low Energy

In this section, we introduce the most important Bluetooth Low Energy concepts defined in the Bluetooth specification [1]. This background knowledge is fundamental for understanding the subsequent sections. In principle, there are two methods that BLE devices can use for communication: using connectionless *broadcasting*, or by establishing *connections*.

Broadcasting

BLE advertising packets are used to broadcast data to multiple peers at the same time. Other devices can run a *scanning* procedure to read advertisement packets. Each advertisement packet can carry 31-byte payload to describe its capabilities and or any other custom information. Optionally, the scanning device can request a *scan response* from the advertiser, in which the advertiser can send an additional 31-byte payload. That means 62 bytes in total can be transmitted using the broadcasting mechanism. It is important to note that this is unidirectional data transfer and the broadcaster has no way to specify who can receive those packets, or receive any acknowledgements.

Connections

An advertising packet can be marked as *connectable*. In that case, if data have to be transferred bidirectionally or more than 62 bytes are required, a connection between two devices can be established.

Devices in BLE can act in two roles: *centrals* and *peripherals*. A central repeatedly scans for advertising packets broadcasted by peripherals and when needed, it initiates a connection. A peripheral then periodically broadcasts advertising packets and accepts incoming connections. There are no restrictions on connection limits imposed by the specification. Since Bluetooth 4.1, a single device can act both as a central and peripheral at the same time, and it can also be connected to multiple centrals/peripherals.

Address Types

The BLE protocol stack differentiates between two types of addresses. The *public address* is a standard IEEE-assigned MAC address that uniquely identifies the hardware device. Since all BLE packets include a device address, it would be possible to track device movement by adversial scanners. BLE addresses this issue by using a *random address* for any communication. This address is generated using the combination of a device *identity resolving key* and a random number, and it can be changed often, even during the lifetime of a connection.

4.6.2. BLE Communication Architecture

As mentioned earlier, the primary purpose of BLE was to enable exchange of information with peripheral devices. However, as it is currently the most universal way for nearby communication on mobile devices that does not require any user interaction, it is potentially suitable for any type of communication. Since Android 5.0, it is possible to create a custom GATT server which allows two Android devices to communicate with each other over BLE. We now proceed to designing a system architecture for P2P communication using BLE. The overall high-level architecture of data flow is shown in Figure 4.4.

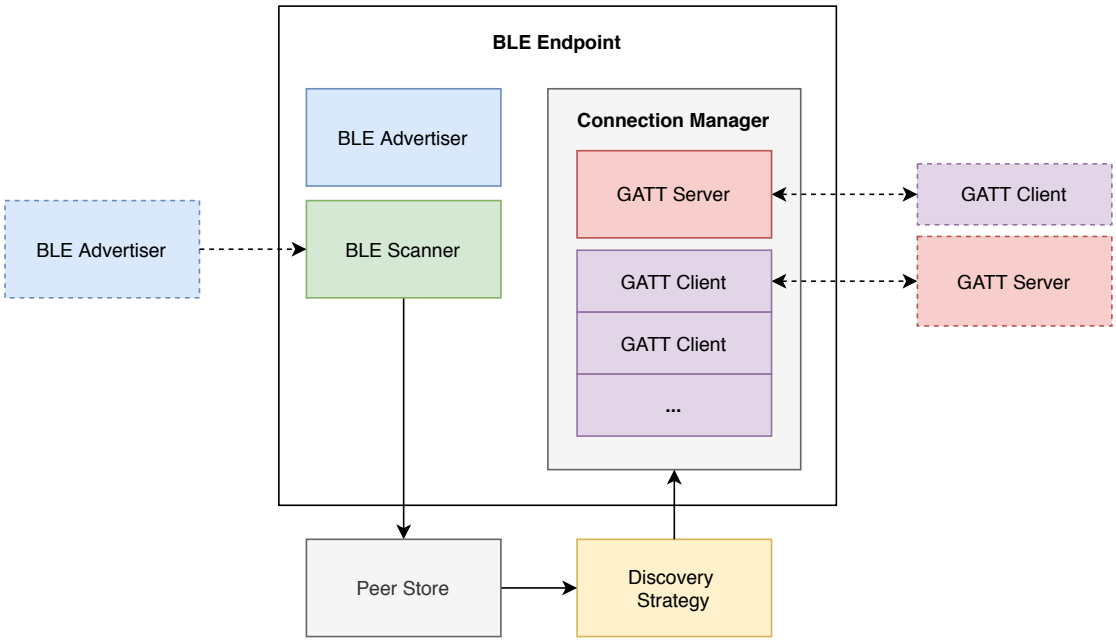

Figure 4.4: Our Bluetooth Low Energy Communication Architecture

The BLE module should be composed of several submodules with clearly separated responsibilities. The communication begins by the device A broadcasting connectable advertising packets using **BLE Advertiser**. The advertising packet contains:

- a *service UUID* which identifies our application,
- a *transmission power level* in dB which can be used by the receiver to calculate a path loss and estimate the distance between devices, and
- a *peer ID* which identifies the broadcasting device.

The device B then scans for advertising packets using a **BLE Scanner**. It should filter packets by service UUID to receive only packets relevant to our application. The BLE scan is a power-intensive operation, so it should be performed only when the user actually wants to connect to a new device. It could be done e.g. only when the application is in the foreground. In case we are designing a long-running service that should run in the background, a scan should be run periodically. We should have an option to specify a scan window duration and an interval between individual scans.

Once the scanner receives an advertising packet, it creates a *peer candidate* which consists of a peer ID, a Bluetooth device address which can be used to initiate a connection, a transmission power level in dB, and a received signal strength (RSSI) in dBm. It stores the peer candidate into the **Peer Store**.

The **Discovery Strategy** is responsible for selecting which peer we should connect to. The strategy should be application-specific and can e.g. prefer to connect to devices with the largest RSSI value, or to connect only to known peers based on their peer ID. Once the strategy selects a peer it wants to contact, it is sent to the **Connection Manager**.

The connection manager contains all GATT-related communication logic. Each device has exactly one GATT server and an arbitrary number of GATT client instances, one for each

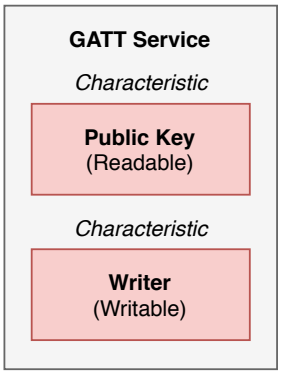

Figure 4.5: Our GATT Server Architecture

device it is connected to. The GATT server implements a single GATT service containing two characteristics which are shown in Figure 4.5. The **Public Key** characteristic has a *readable* permission and simply contains a public key of the peer. It is used to determine the identity of the device when initializing the connection and can be used for authentication and encryption in the further communication. The **Writer** characteristic with a *writable* permission is then used for sending data from the client to the server. As we want to support bidirectional communication, every two devices need to have a pair of client–server and server–client connections. This can be implemented in a way that every time a GATT server receives an incoming connection, the connection manager initiates an outgoing connection to the GATT server of the connecting device. Only after both links are established, the connection is considered ready.

4.7. TrustChain: Scalable Distributed Ledger

Any service that allows interaction between users needs a way to record and exchange data. In traditional services, this is done using a client-server model where a server is the source of truth. In a distributed system where data need to be synchronized across many independent nodes, a lot of problems arise.

In our stack, we use TrustChain [21], a graph-based append-only datastructure. It was designed to record any type of transactions between strangers without central control. Compared to traditional blockchains such as Bitcoin or Ethereum, it does not require global consensus and thus can work in partitioned networks and is linearly scalable.

In TrustChain, every user has its own chain with its own *genesis* block. Every block contains exactly one transaction. Whenever a transaction is created between two parties, a block is added to the chains of both participants, which causes *entanglement*.

A block is usually composed of two half-blocks, where a half-block is added to the chain of each transaction participant. For this reason, transaction creation needs to be interactive. Let's assume A wants to perform a transaction with B. First, A creates a *proposal block* containing its public key, the sequence number of the previous block, the hash of the previous block, and the public key of the conterparty (a *link public key*). Finally, A signs the *proposal block* with its own private key and sends it to B. Upon receiving the proposal block, B validates its integrity. If the block is valid and B agrees with the proposed transaction, it creates an *agreement block*. The agreement block should contain the public key of B, the sequence number of its previous block, a public key of A, and the sequence number of the other half-block in the chain of A. Finaly, B signs the agreement block with its private key and sends it back to A. After this, the transaction is considered completed.

5

Protocol Implementation

In this section, we describe the system architecture and implementation of the P2P communication library. One of our main goals is to create an implementation that would be compatible with the majority of mobile devices. The most commonly used operating systems are Android and iOS, where Android uses the Android Runtime, with primary programming language being Java or Kotlin. iOS runs ARM native code compiled from Objective-C or Swift. Both platforms support running native code and allow the use of bridges to access system APIs. We are primarily interested in supporting Android, but extension for iOS support should be possible in the future.

Regarding the programming language choice, many options have been considered. In general, there are three approaches to mobile development. The first one is to write the core logic in a language that compiles to native code and compile it as library for all possible CPU architectures. *Rust*[1] was considered as a low-level language that compiles to native code. It provides good performance and its compiler is proven to prevent any race conditions in a multi-threaded code. However, it could only be used to develop the core library, and the UI would still need to be written in languages supported by the specific platform SDK.

Another approach is using a multi-platform framework that allows to reuse most of the code including the UI. Probably the most prevalent multi-platform framework today is *React Native*[2] developed by Facebook. However, it comes with the overhead of running in the JavaScript engine and despite its name, does not really provide a native performance.

The last and the most common approach is to use the languages officialy supported by the platform SDK. *Kotlin*[3] is a modern statically typed language that can be compiled into JVM, native code, or JavaScript. It has been officially supported as the official programming language for Android since 2017 [5] and has replaced Java as the primary language in 2019. Thanks to its ability to compile to native code using the Kotlin/Native compiler, it can also be compiled for other platforms such as iOS. In the end, We have chosen Kotlin as the language for implementing both the P2P library and eventually the application itself.

5.1. System Architecture

The architecture of kotlin-ipv8 is heavily inspired by the architecture of the original py-ipv8 library, with some extensions. The basic architectural building blocks are visualized in Figure 5.1.

[1] https://www.rust-lang.org/
[2] https://reactnative.dev/
[3] https://kotlinlang.org/

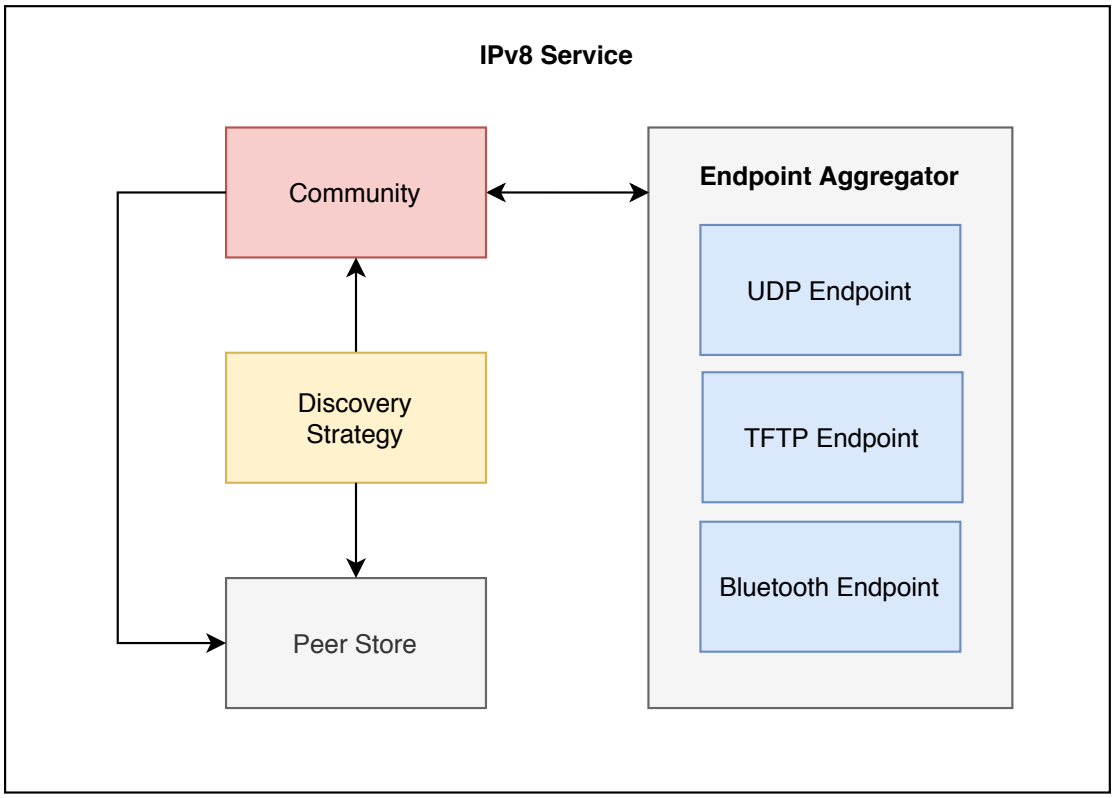

Figure 5.1: The system architecture of the IPv8 service

A *community* is a service implemented by the developer, or provided by the core of the library. Every service using the library should be extending the base `Community` class which provides convenient methods for serializing and deserializing messages, and reading the list of peers who are present in the community from the *peer store*. It also implements messaging required for peer introduction and NAT puncturing mechanism previously described in Section 4.4.

Every community can have assigned one or more *dicovery strategies*. A discovery strategy is primarily responsible for actively discovering and connecting to new peers in the network who are present in the same community. The discovery strategy should implement a method `takeStep` that is called reguarly in a preconfigured step interval and performs the required task.

Finally, an *endpoint* is an service facilitating communication with the outside world. It is responsible for opening a communication socket, and sending and receiving serialized packets from other peers over the opened socket. The library supports multiple transports implemented by separate endpoints. To abstract this away from the application developer, the community only communicates with the *endpoint aggregator*. It groups all endpoints and decides to which endpoint the message should be sent. The messages received by the endpoint are then forwarded to the correct community based on the message header.

5.2. Communities

Each service has to extend the abstract `Community` class which implements the `Overlay` interface. The only field left for the developer to define is `serviceId`. This can be an arbitrary 20-byte array represented as a hexadecimal string that uniquely identifies the community. The Community class defines several methods that can be used by its subclasses:

- `getPeers(): List<Peer>` – Returns the list of connected peers that are present in this community.

- `serializePacket(messageId: Int, payload: Serializable, sign: Boolean, encrypt: Boolean, recipient: Peer?): ByteArray` – Serializes a payload into a binary packet that can be sent over the transport. The method can also handle signing or encrypting the serialized packet data. For encryption, a `recipient` parameter needs to be provided.

- `send(peer: Peer, data: ByteArray)` – Sends a packet to the target peer.

- `send(address: Address, data: ByteArray)` – Sends a packet to the target IPv4 or Bluetooth address.

5.2.1. Discovery Community

`DiscoveryCommunity` is one of the core communities implemented by the IPv8 core. It tries to keep an active connection with a specified number of peers and keeps track of communities they participate in by sending *similarity requests*. A similarity requests contains a list of communities the sender supports, and expects to receive a *similarity response* back, which will include the list of communities supported by the other side. Furthermore, it implements *ping* and *pong* messages that allow to perform keep-alive checks, measure latency between peers, and handle peer churn by removing churned peers from the peer store. While it is possible to run IPv8 without using this community, it is not recommended. The discovery community is expected to be used with `PeriodicSimilarity` and `RandomChurn` strategies.

5.2.2. TrustChain Community

`TrustChainCommunity` implements TrustChain, a scalable distributed ledger previously described in Section 4.7. The community implements 3 use basic cases: interactive block signing, broadcasting, and crawling.

Whenever a new proposal block is created, it is sent using a *half block* message to the counterparty stated in the link public key field. The counterparty should create and sign an agreement block and send it back in another half block message. The proposer should wait for the response and re-send the half block to the counterparty if they do not respond within a timeout interval.

Whenever a peer signs a proposal or an agreement block, they should broadcast it using a *half block broadcast* or *half block pair broadcast* message respectively. Any peer who receives the broadcast message should decrement TTL and re-broadcast it.

The *crawl request* is used to retrieve a chain of blocks with a specific public key. The start and end sequence numbers should be specified to request only the part of the chain we don't have yet. A unique crawl ID should be included in each crawl request. After initiating a crawl, the initiator should wait until they receive all blocks from the requested range. If all blocks are not received within a timeout interval, a new crawl request should be sent with the start sequence number updated. Whenever a peer receives the crawl request, it should send all blocks within the specified range in separate *crawl response* messages, or send an *empty crawl response* if there is no block in the specified range.

5.3. Discovery Strategies

A discovery strategy is primarily responsible for actively discovering and connecting to new peers in the network who are present in the same community. There are a few strategies with

different use cases implemented by the IPv8 core, but developers are free to implement their own strategies suitable to their needs.

5.3.1. Random Walk

`RandomWalk` is a simple discovery strategy that performs a random walk in the network. On every step, it requests a new peer introduction by sending an introduction request to a random connected peer. At the same time, a random peer from peer candidates (the list of introduced, but not yet contacted peers) is selected, and greeted with an introduction request. If the peer does not respond with the introduction response within a specified timeout (3 seconds by default), it is removed from the peer candidate list.

5.3.2. Periodic Similarity

The `PeriodicSimilarity` strategy simply sends a similarity request on every step to a random peer in the peer store, and thus ensures the list of communities for every peer is kept up to date. It is only intended to be used with `DiscoveryCommunity`.

5.3.3. Random Churn

The `RandomChurn` strategy starts by selecting a list of random peers on every step. Each peer in the list is classified either as *active*, *inactive*, or *churned*. The peer is considered churned if we haven't received a response to a ping in the last 57.5 seconds. In that case, we drop the peer from the peer store. The peer is considered inactive if we haven't received a response in the last 27.5 seconds. In that case, we send a ping to check for liveliness. If the peer is still active, we only send a ping if we haven't send it yet, to calculate the latency. The magic constants for detecting inactive and churned peers have been adopted from the original py-ipv8 implementation, as they have been extensively tested in the wild and are expected to reflect the behavior of the majority of NAT boxes. [12]

5.3.4. Bluetooth Discovery Strategy

`BluetoothLeStrategy` is a default strategy for initiating connections with nearby peers. In each step, it queries all discovered but unconnected peers, and sends a connection request to the peer with the highest RSSI value. It limits the maximum number of peers to 7, as it is known to be the the highest amount of peers Android Bluetooth implementation is able to handle reliably. For more dense nearby communication, a mesh network should be built on top of the existing protocol.

5.4. Bootstrap Server

The bootstrap server is used by new peers who are interested in joining the network. It is implemented by extending the base community class with a customized behavior. The server accepts and responds to introduction requests regardless on the community id sent in the header, which enures that the bootstrap server can be used for discovering peers in any community. Upon the introduction request, it adds a peer to its peer store, and responds with an introduction response that contains the link IP address of the peer as it was received on the socket, and a random peer from the community from which the request was send. Peer churn is handled by a `SimpleChurn` strategy that simply removes any peers that have not sent any message in the last 120 seconds.

The bootstrap server can be started with the following command, where a port can be specified in the `port` Java property:

```
./gradlew :tracker:run -Dport=8090
```

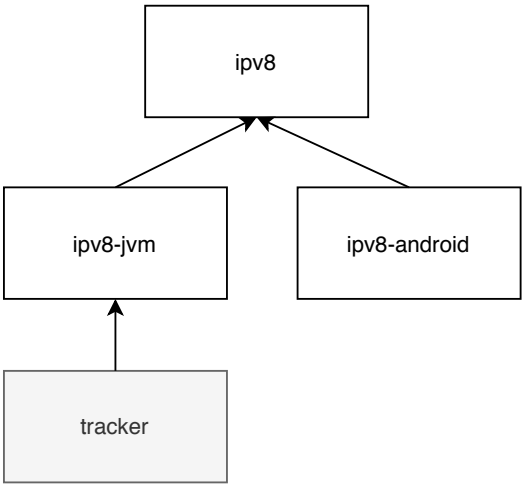

Figure 5.2: The dependency graph of kotlin-ipv8 library modules

The list of default bootstrap servers that are contacted upon initialization is defined in the `Community.DEFAULT_ADDRESSES` field.

5.5. Project Structure

The project consists of two sub-projects that are maintained in two separate repositories: the P2P library called *kotlin-ipv8*[4], and the application built on top of this library. The kotlin-ipv8 library currently supports Android and JVM targets, and the project is composed of the following modules:

- `ipv8` *(JVM library)*: The core of the IPv8 implementation, a pure Kotlin library module.
- `ipv8-android` *(Android library)*: Android-specific dependencies and helper classes for running IPv8 in Android Runtime.
- `demo-android` *(Android app)*: An Android application demonstrating the initialization of the `ipv8-android` library.
- `ipv8-jvm` *(JVM library)*: JVM-specific dependencies for running IPv8 on JVM.
- `demo-jvm` *(JVM app)*: The CLI app demonstrating the usage of the `ipv8-jvm` library.
- `tracker` *(JVM app)*: The bootstrap server implementation.

The ipv8 module contains the core logic that is shared across JVM and Android platforms. To ensure the stability of the project and make contributions by other developers easier, a development and *continuous integration (CI)* infrastructure has been set up. The tests are written using the test framework *JUnit*[5] and mocking framework *mockk*[6]. The code coverage is calculated and reported using *Java Code Coverage Library (JaCoCo)*. The *ktlint*[7] linter is used to ensure the consistent code style, and Android Lint to check for common errors related to the usage of Android SDK. All tools are run automatically for every commit and merge request by a *GitHub Actions* workflow. *codecov.io*[8] is used to report the changes in code coverage for every pull request.

[4] https://github.com/Tribler/kotlin-ipv8
[5] https://junit.org/
[6] https://mockk.io/
[7] https://ktlint.github.io
[8] https://codecov.io

6

Decentralized Super App

Inspired by the emerging concept of super apps, we propose our own variant of a super app which builds on top of our decentralized stack, and is not dependent on any centralized infrastructure. The screenshot of the launcher screen with the list of available mini-apps is shown in Figure 6.1.

All mini-apps are currently implemented in a separate repository *trustchain-superapp*[1] and include kotlin-ipv8 as a Git submodule. In the future, we envision a decentralized app store, where users would be able to publish, download, and update mini-apps created by the community, without relying on centralized app stores. This idea is highly related to the previous research on FBase [20]. However, its implementation would need further research in dynamic code execution in the context of Android Runtime, and is out of scope of this thesis.

6.1. PeerSocial: Decentralized Social Network

In our initial implementation, we focus on providing basic building blocks for a decentralized social network that should in principle be able to replace services provided by Facebook and WeChat. There are three basic funcionalities we want our social network to provide:

- A friend list with secure key exchange mechanism that prevents impersonation and man-in-the-middle attack
- End-to-end encrypted private messaging
- Public post feed with likes and replies

6.1.1. Trustworthy Friendship Establishment

Before any communication starts, it is important to know who we are communicating with. In traditional social networks, users are simply identified by their name, e-mail, or a phone number. However, relying on these attributes makes it relatively easy to perform an impersonation attack, a form of attack in which an adversary acts as a trusted person with the aim of financial or intellectual loss.

There are several ways of establishing trust in cryptography systems, with different compromises between security and convenience. Many commonly used protocols have adopted *Transport Layer Security (TLS)* protocol to secure all communication between clients and servers. The trust in TLS is enforced by using *public key infrastructure (PKI)* with trusted certificate authorities. As no central authority is desired in peer-to-peer systems, there is a

[1] https://github.com/Tribler/trustchain-superapp

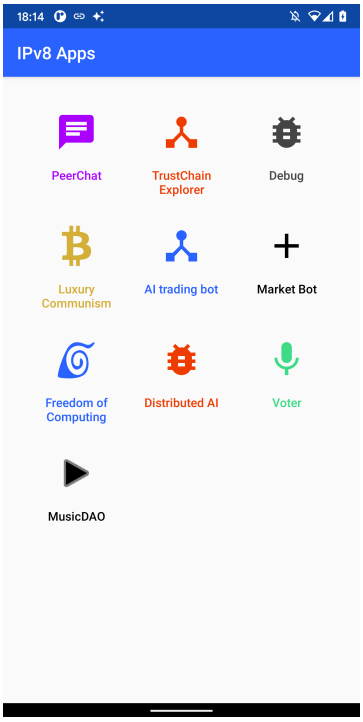

Figure 6.1: The dashboard of the super app composed of several decentralized mini-apps implemented on top of the kotlin-ipv8 library and TrustChain.

decentralized alternative called a *web of trust*, which is characteristic for systems based on *Pretty Good Privacy (PGP)*. In the web of trust, identity certificates (certificate connecting public keys to owner identity) are signed by its users. This is usually done in person at key signing parties. A weaker form of security is provided by *trust on first use (TOFU) authentication* scheme, where the link between the identity and the public key is established during the first communication touchpoint. After that, the public key is stored in the local trust store, so any man-in-the-middle attack trying to change the public key is detected.

In our system, we require users to exchange public keys before they start communicating. The key exchange can happen either in person, or over a secure channel. The implemented user interface for secure key exchange mechanism is shown in Figure 6.2. When both contacts are in proximity, they can simply scan each other's QR code which encodes the public keys. This is the most secure way of exchanging keys, as users can be sure the key has not been tampered with. When the users want to establish communication remotely, they can exchange the public keys in hexadecimal encoding over a secure channel. In this way, the security of the key exchange relies on the security of the communication channel. Finally, the user is prompted to assign a human-readable name to the public key of the other party.

After saving the contact, it is shown in the contact list with its name, public key, and the text and timestamp of the last exchanged message. The contact list UI is shown in Figure 6.3.

6.1.2. Private Messaging Protocol

We now build on top of our identity and networking layer to provide a decentralized alternative of a messaging application. It should allow users to send end-to-end encrypted private messages. In addition to text messages, it should also allow to transfer images and other binary files.

We design a simple messaging protocol consisting of two payload types: a *message*, and an *acknowledgement*. The message payload contains a unique id, a text message, and an

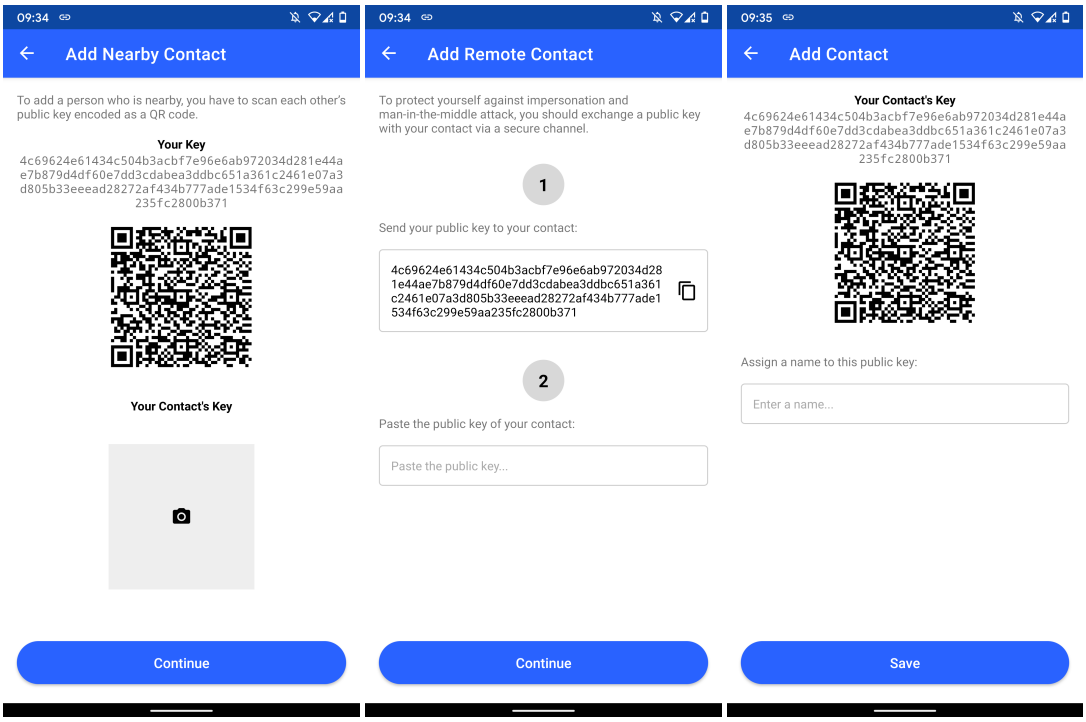

Figure 6.2: The UI for adding a public key of a new nearby or remote contact to the trusted public key store, and associating the public key with a human-readable contact name.

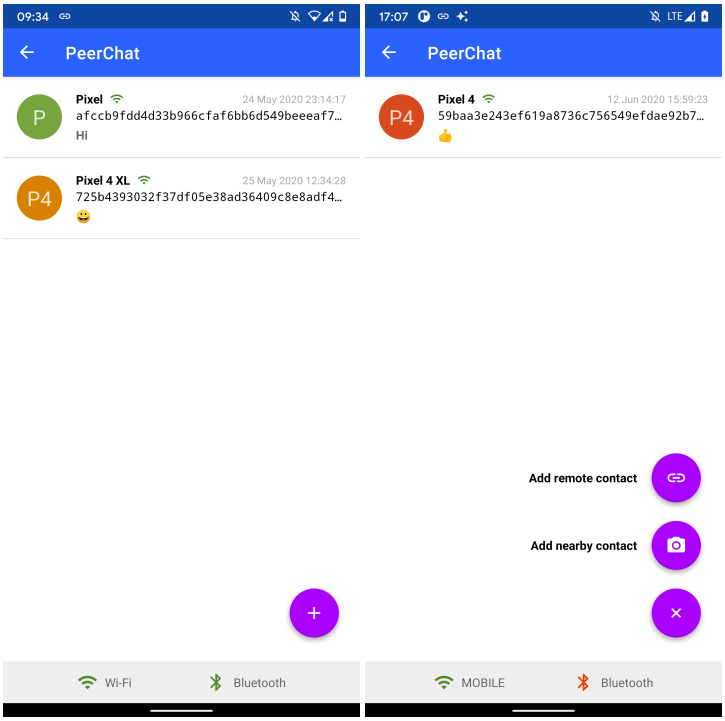

Figure 6.3: The list of contacts with online indicators, their latest messages with timestamps, and the connectivity status of the local UDP and Bluetooth endpoints.

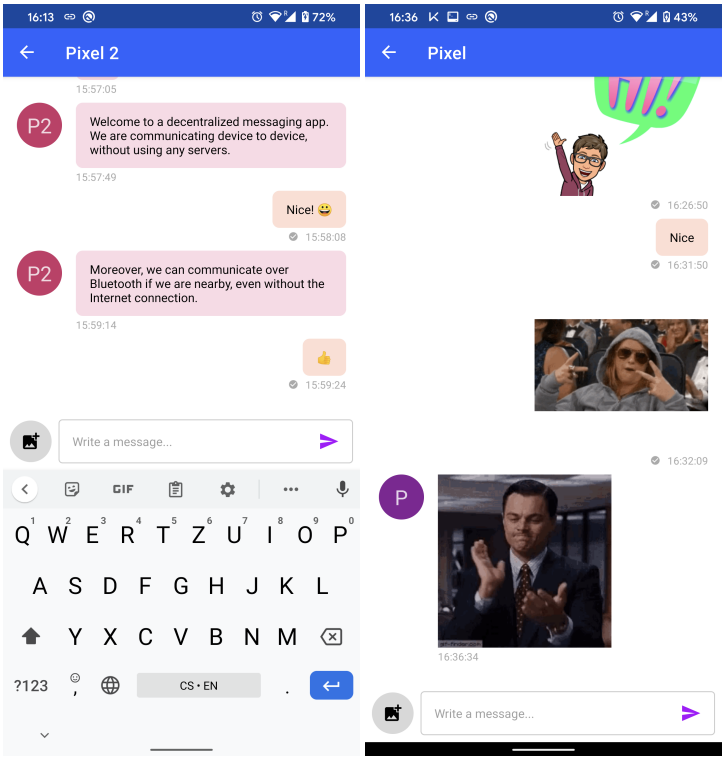

Figure 6.4: The conversation UI containing incoming and outgoing text messages, image attachments, and delivery indications for outgoing messages.

optional attachment metadata consisting of the attachment type, size, and id. The acknowledgement payload contains simply the id of the message whose delivery should be acknowledged.

The protocol works as follows. When the user wants to send a message, it sends a message payload and awaits to receive the acknowledgement from the recipient. If the acknowledgement is not received within a timeout, the message is being periodically resent and the operation is repeated. This ensures that all messages are eventually delivered when both peers are connected.

As attachments can be arbitrarly large, we want to leave it up to the receiver to decide when they should be fetched from the sending device. For example, the user might not want to automatically fetch large attachments over a metered LTE connection which has an associated data limit. For this reason, the message payload includes only the attachment ID which can be used to request the content of the attachment. The attachment ID is the hash of the content, which effectively prevents file duplication when a single file is shared by multiple people.

We define two more payloads to support attachments: an *attachment request*, and an *attachment response*. The attachment request contains only the attachment ID and is used to request the content of the attachment. When a peer receives an attachment request, it should check if the requested attachment is stored in its storage and respond with the corresponding attachment response containing the binary attachment data.

All messages are signed with the private key of the sender and encrypted with the public key of the recipient, using the built-in functionality of the kotlin-ipv8 library. The screenshots of the conversation UI showing text messages, attachments, and delivery indicators, are shown in Figure 6.4.

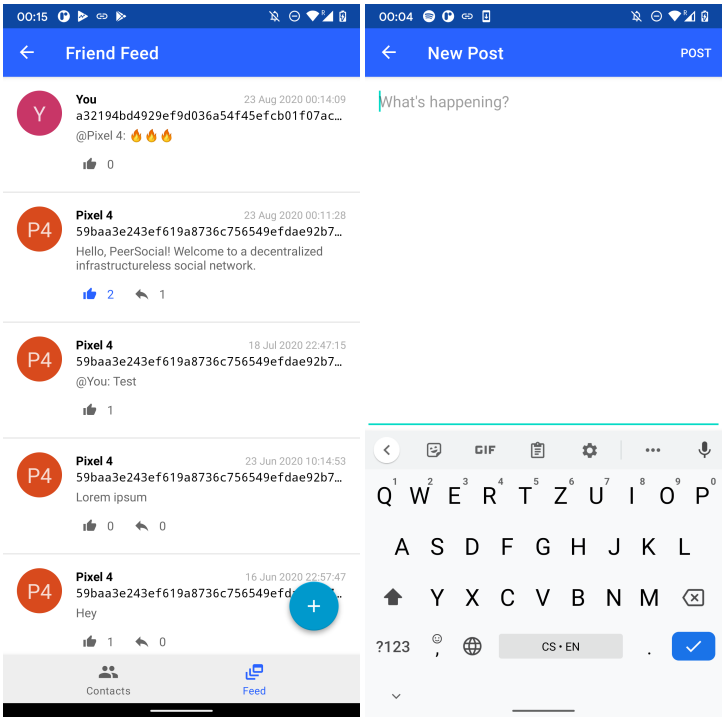

Figure 6.5: The social feed with likes and replies and the screen for composing a new post.

6.1.3. Public Post Feed

To complete our idea of a social network, we need to add a way for users to share information publicly. Each user should have their own feed in which they can publish posts. Other users should then be able to interact with those posts by publishing replies or signal agreement by liking those posts.

We take advantage of the TrustChain ledger as a data storage and synchronization mechanism. Its acyclic graph data structure is perfectly suited for our use case of modelling a social network. The feed is represented by blocks in the user's chain. There are two types of blocks representing posts and likes. A post is created as a proposal block with `ANY_COUNTERPARTY` constant used for the link public key, which means anyone can create an agreement block. The agreement blocks then represent replies to the linked post. The agreement block can be linked to a post, or also to another reply. This will ultimately result in multiple agreement blocks linked to a single proposal block if there are more than a single reply or like for a given post.

In our prototype, posts can only contain text messages, but the functionality could be easily extended to also support binary attachments, using the same content-addressable mechanism to include and fetch attachments as previously described in the private messaging protocol.

Our data storage acts as an append-only ledger, which has an interesting consequence that anything that has been once published cannot be changed or removed, as the history is immutable. If edit or remove functionality was desired in the UI, it could be implemented as another type of block changing the state of the original block. The application would then need to replay all blocks in the chain to determine the final state of the feed and find out about possible post updates or removals.

The UI of our initial social network implementation is shown in Figure 6.5, displaying the social feed of friends with likes and reply count.

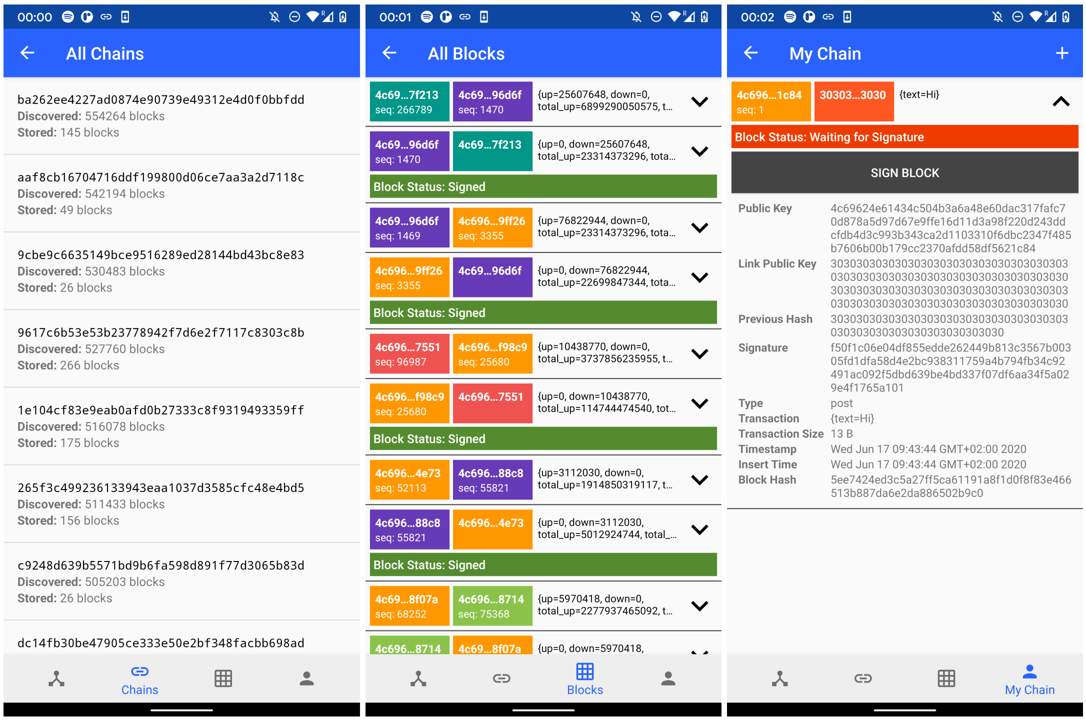

Figure 6.6: The list of all chains, blocks, and a block detail in the TrustChain Explorer app.

6.2. TrustChain Explorer

TrustChain Explorer is a tool that allows to browse the blocks received from the network and stored in the local store. Its main purposes are to simplify application development and debugging, and to familiarize new developers with the main concepts used in TrustChain. The main parts of the UI can be seen in Figure 6.6. The first tab shows the list of chains ordered by the number of discovered blocks. After clicking on the chain, all blocks stored for the given chain are shown. Each block can then be expanded to show all information stored in the block, including public keys and decoded transaction. In the second tab, all recent blocks are shown, sorted by the insert time. The list is refreshed automatically, so it looks like a live stream of incoming blocks. In the last tab, blocks in the chain of the current user are shown. The user can create a new proposal block with an arbitrary transaction content by clicking on the plus button. Finally, they can counter-sign incoming blocks that are awaiting their signature. The TrustChain Explorer application is not very useful on its own, but it has proved to be valuable during development of applications using the TrustChain ledger.

6.3. Project Structure

From the implementation perspective, the whole super app is composed of several application modules. In general, each mini-app is implemented in a separate module, independent from other mini-apps. The code that needs to be shared among all apps is present in the `common` module, which all other modules are dependent on. Finally, the dashboard with the list of available apps and links to them is implemented in the app module. The whole dependency graph is visualized in Figure 6.7. Overall, the application contains 5888 lines of code (only the source code excluding empty lines and comments). The breakdown by application module is listed in Table 6.1.

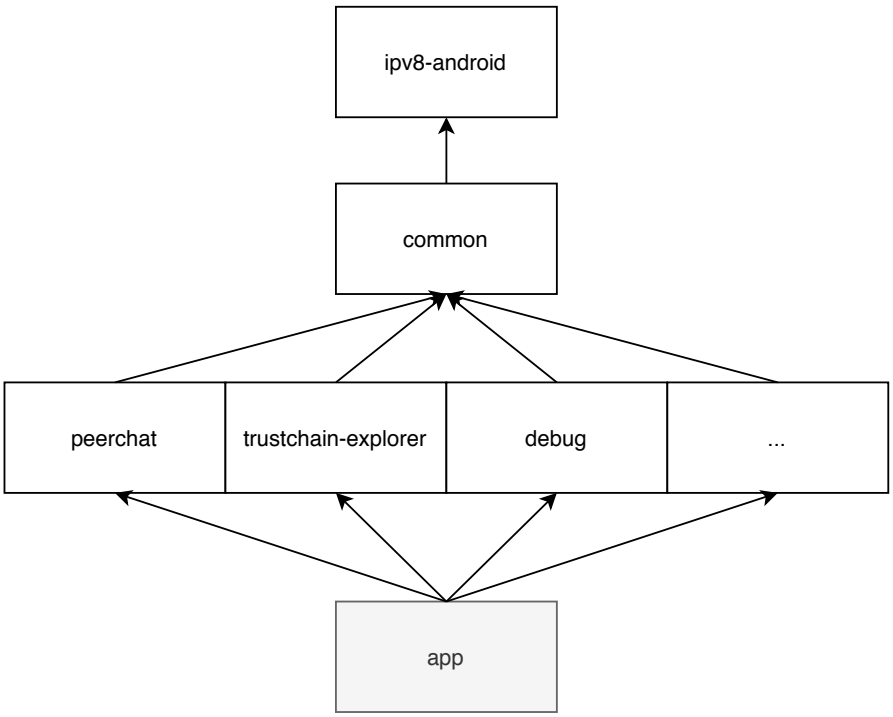

Figure 6.7: The dependency graph of the superapp modules

Module	Lines of code
common	847
app	560
debug	542
peerchat	2701
trustchain-explorer	1238
	5888

Table 6.1: The lines of code by superapp module breakdown.

7

Evaluation

In this chapter, we perform an experimental evaluation of various components of the developed networking library, to assess its scalability and production-readiness. The lack of end to end connectivity is a big problem in P2P, so we focus on that first in our evaluation. We analyze the success rate of the NAT traversal mechanism on carrier networks of all major internet service providers (ISPs). Then, we perform a stress test by connecting to up to 500 peers in the live network. During the experiment, we observe the message complexity and the ping latency of a given peer while the number of connected peers is increasing. We measure and evaluate the transfer rate of the binary file transfer protocol. Subsequently, we analyze the bootstrap performance to find out how long does it take to become fully connected to the network from the initial bootstrap server contact. We analyze the power efficiency by measuring how much the application affects the battery life of the device. Finally, we evaluate the code quality of the project from the test coverage perspective.

7.1. Analysis and Puncturing of Carrier Grade NAT

According to the report by Statista [28], there were three major mobile phone operators providing services in the Netherlands in Q4 2018, as listed in Table 7.1. In total, these represent up to 85% of the market share. The rest of the market is shared by Mobile Virtual Network Operators who sell services over existing networks of those three operators.

We have purchased pre-paid SIM cards for all three major mobile network operators to investigate whether they are suitable for peer-to-peer communication. First, we tried to infer the characteristics of their Carrier Grade NAT deployments.

We used the STUN protocol and NAT behavior discovery mechanisms described in [18]. They have shown that all networks appear to use *Endpoint-Independent Mapping (EIM)* and *Address and Port-Dependent Filtering* (also known as *port-restricted cone NAT*). EIM is a sufficient condition for our NAT traversal mechanism to be successful, so this would make all these NATs suitable for P2P communication.

Operator	Market share
KPN	35%
Vodafone	25%
Mobile Virtual Network Operators	25%
T-Mobile	20%

Table 7.1: Market share of mobile network operators in the Netherlands in Q4 2018. The shares do not sum up to 100% as they are rounded up within five percent ranges in the original report. [28]

However, we also observed that NAT behavior of CGN can change over time, so STUN behavior discovery mechanism is not sufficient to correctly classify CGN behavior. We performed some more tests to verify if the behavior is consistent over time. We attempted to connect to 50 different peers over the interval of 5 minutes. We verified that KPN and T-Mobile networks are consistent with EIM behavior. However, the CGNAT used by Vodafone changes the port mapping for new connections approximately every 60 seconds, even when connecting to the same IP address and a different port. While not strictly correct due to temporal dependency, this behavior could be classified as *Address and Port-Dependent Mapping (APDM)* which is characteristic for a *symmetric NAT*.

The mapped ports seem to be assigned at random, which makes it infeasible to use any known symmetric NAT traversal techniques such as port prediction [29]. However, our symmetric NAT puncturing approach previously described in Section 4.5 has shown promising results which are presented in this section.

7.1.1. Experimental Setup

Our experimental setup for the connectivity test consists of 5 devices connected to different networks. The network topology is shown in Figure 7.1. There are three devices connected over LTE to different mobile ISP networks (T-Mobile, KPN, Vodafone). Then, there are two devices connected to two different home broadband networks provided by Tele2 and T-Mobile Thuis. In this setup, we could test all possible scenarios. In the final phase of the test, some of the devices were reconnected to different networks to test connection between two devices on the same network. For that, an additional SIM card for each of the ISPs was used. Note that when two devices are connected to the same home router, they are effectively communicating only over LAN, but the results were included anyway.

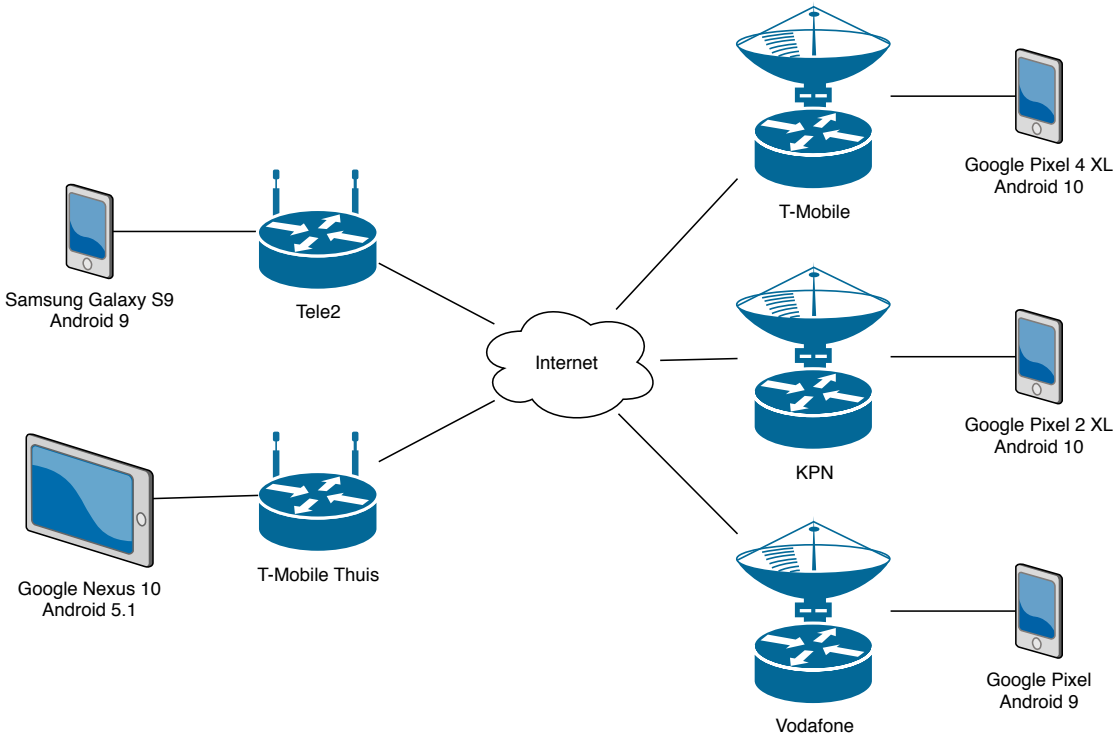

Figure 7.1: The experimental setup for the connectivity test using devices connected to different mobile (T-Mobile, KPN, Vodafone) and home broadband (Tele2, T-Mobile Thuis) ISP networks.

7.1.2. Results

The results of the connectivity test are presented in Table 7.2. We can see that the connection could be established in all of the cases. In most of the cases, for networks that show EIM behavior, a simple UDP hole punching was sufficient to perform the NAT traversal. However, the issues arised with the Vodafone network. In that case, a more advanced multihole punching needed to be employed, which allows the connection to be established between two devices on the Vodafone network. However, the brute-force nature of the mechanism probably exceeded the session limit on the T-Mobile network. It has been experimentally discovered that the limit of open UDP sessions is around 1000 and the binding lifetime is around 30 s. When the puncture rate has been slowed down to 1000 ports per 30 s, the puncturing was successful in all experiment runs, however it can theoretically take up to 30 minutes to discover the mapped port.

	T-Mobile	**KPN**	**Vodafone**	**Tele2**	**T-Mobile Thuis**
T-Mobile					
KPN					
Vodafone					
Tele2					
T-Mobile Thuis					

Table 7.2: The connectivity matrix representing the NAT traversal method needed to establish connection between devices connected via different ISPs (green: hole punching, yellow: multihole punching, orange: delayed multihole punching).

7.2. Social Graph Scalability Experiment

The average Facebook user has 155 friends [9]. We perform a stress test by connecting to up to 500 peers and measure how our system handles the increased communication load, to show that our solution scales well beyond the average use.

First, we measure the message complexity as the number of messages received from other peers per minute. The messages mostly consist of introduction and similarity requests and responses sent by TrustChain and discovery community, but they also also include blocks broadcasted by connected peers. In Figure 7.2 we can see that the number of messages scales linearly with the number of connected peers.

During the whole test, we have also measured the latency by sending pings from another device to the device under test to verify how the response time changes with the increasing number of peers. We can see that the latency remains mostly constant during the whole test, which means the number of peers does not have a significant effect on the performance.

7.3. Bootstrap Performance

For the next two experiments, we take advantage of the existing live IPv8 network, consisting mostly of Tribler users. We run our novel Kotlin implementation of IPv8 on Android and connect to rest of the existing network to evaluate various metrics in scale.

First, we evaluate the bootstrap performance by measuring how long does it take to discover and connect to 30 random peers in the network. We use a random walk discovery strategy which is set to discover 30 peers, and a step interval of 0.5 s, which means new peer introductions are requested at this interval. We run the bootstrap mechanism three times, and reset the IP address before each run to drop any existing connections. The results of all runs are averaged out and visualized in Figure 7.3. It can be seen that the target of 30 peers is on average reached in 35 seconds.

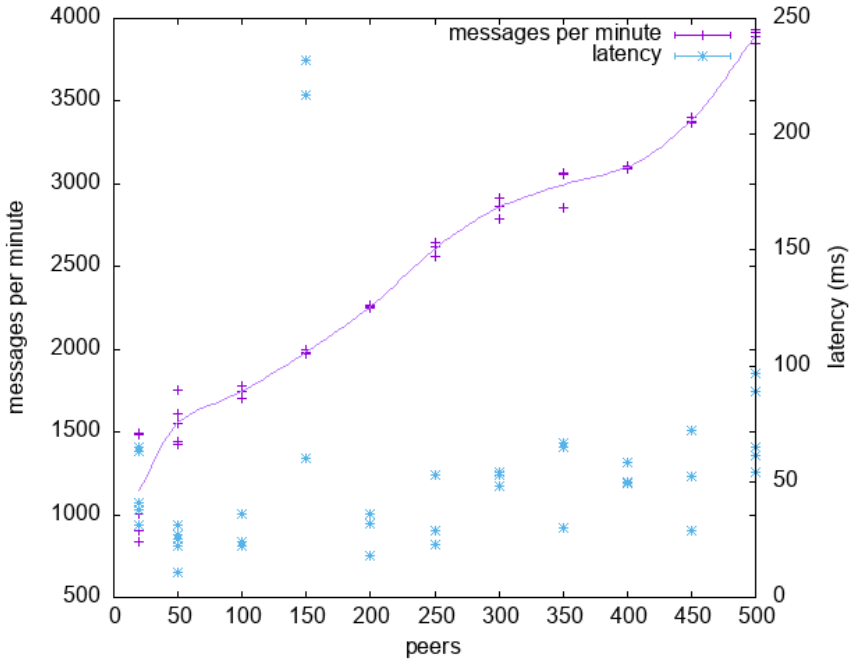

Figure 7.2: The number of messages per second and ping latency based on the number of connected peers.

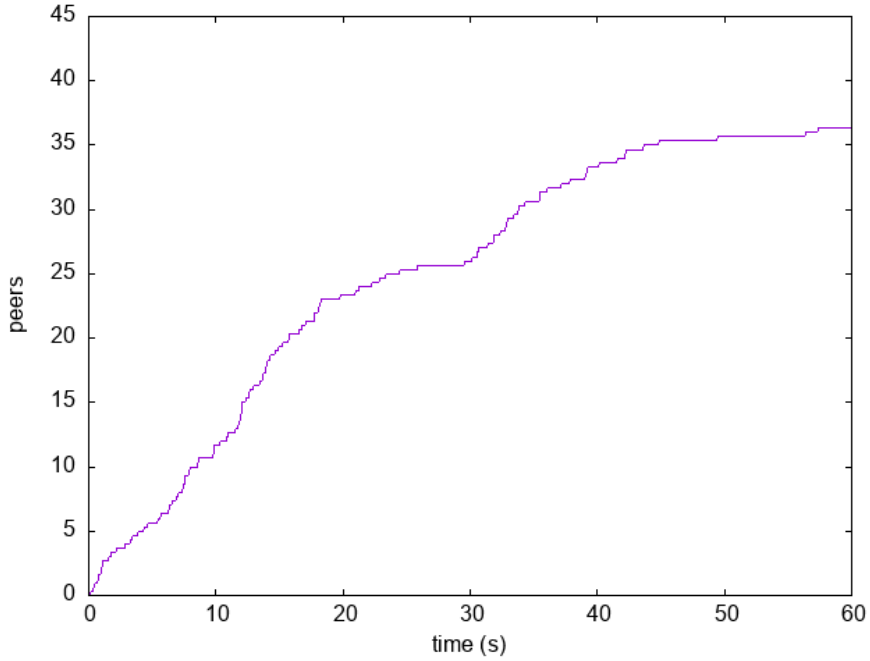

Figure 7.3: The bootstrap performance when using a random walk discovery strategy with a step of 0.5 s. Data averaged out over 3 independent runs.

7.4. Blob Transfer Rate

Our protocol allows to send messages of any size by automatically switching to the TFTP protocol when sending messages larger than UDP packet size. To evaluate this functionality, we measured the transfer rate of the implemented file transfer protocol by sending JPEG images varying in sizes in steps of 50 kB between two Android devices. The resulting average transfer rate is shown in Figure 7.4. Even though the test was performed over LAN on Wi-Fi with

65 Mbps transfer rate capability, the achieved transfer rate is only around 60 kb/s. It is expected for the transfer rate to be much lower than the theoretical upper bound due to the triviality of the protocol. Much higher speeds could be achieved by implementing a more advanced binary transfer protocol based on a sliding window such as uTP or QUIC. However, the transfer rate is stable with the increasing image size, which validates the correctness of the implementation.

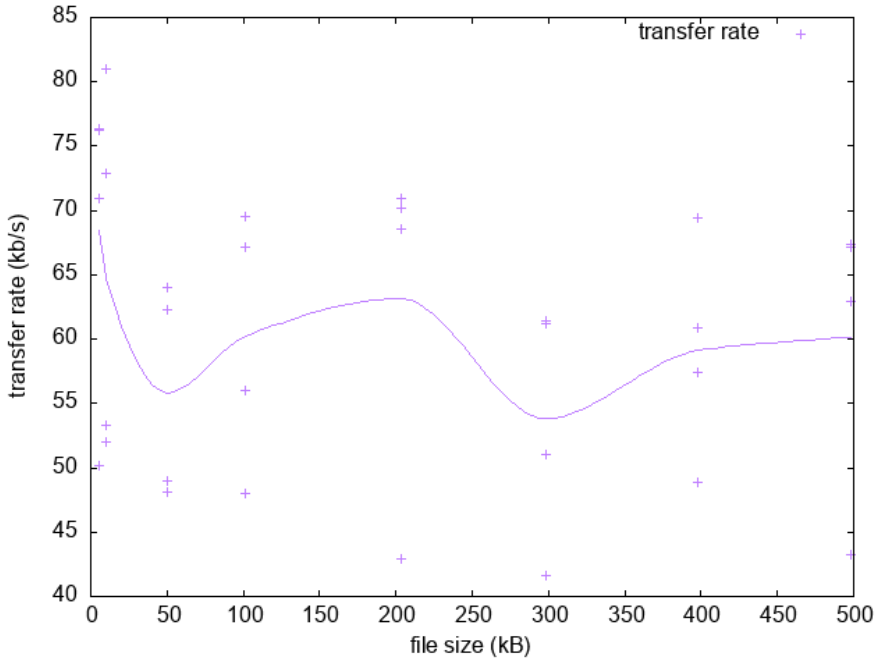

Figure 7.4: The duration and speed of binary file transfer over TFTP based on the file size. The test setup consisted of two devices connected to the same Wi-Fi with 65 Mbps capability.

7.5. Power Efficiency

One of the biggest threats for P2P communication on mobile devices is the battery consumption. To ensure the correct behavior of the application, it is necessary to keep an open long-lived connection with regular keep-alive checks. This has a significant negative effect on the battery life, as it prevents the device from using its power-saving features such as Doze mode. We performed a long-running experiment to measure this effect in practice. The results are shown in Figure 7.5.

We used Google Pixel with Android 9 to capture the battery life behavior in three different scenarios, and conducted each experiment twice. First, we started our application and let it connect to 200 peers. Then we turned off the screen, but kept the application running in the foreground service, allowing it to keep the connections open. In this case, the battery drained quickly in 6 hours. After that, we limited connections to 20 peers. In that case, the battery life was slightly better and lasted for 10 hours. Finally, we left the application turned off. This allowed the device to take advantage of power-saving features and only schedule network requests for background applications in batches. In that case, the battery level was still on 90% after 10 hours.

The experiment shows that it might not be practical to establish long-lived connections and keep the application running in the foreground for longer periods of time. Instead, it might be necessary to only connect to the network when the user interacts with the app, or to occasionally fetch the current state from the network in the background.

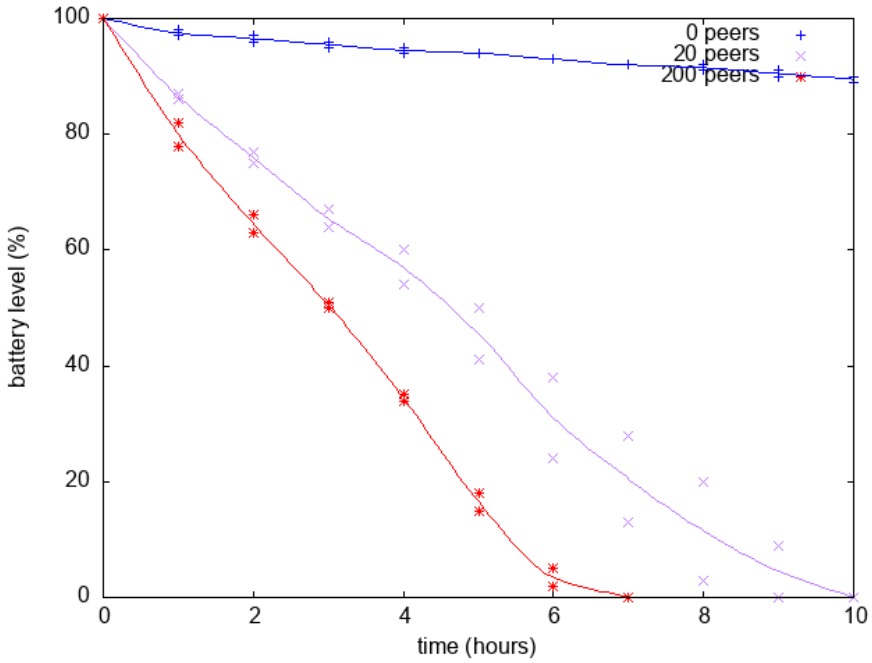

Figure 7.5: The relation between the number of connected peers and the battery usage.

7.6. Code Quality

Finally, we analyze the code quality of the resulting codebase. For a maintainable library, a high code coverage is crucial, as it makes contributions easier and reduces the possibility of introducing new bugs and regressions. Therefore, testability was a major concern during the development of the library. In total, the kotlin-ipv8 library consists of 4134 lines of code (includes only the Kotlin code without comments and empty lines) where 72% of the logic is covered by unit tests. The numbers broken down by packages are shown in Table 7.3.

Package	Lines of code	Code coverage
trustchain	1457	69%
keyvault	177	89%
messaging	1116	67%
peerdiscovery	698	80%
util	68	88%
core	616	77%
	4134	**72%**

Table 7.3: The lines of code and code coverage in kotlin-ipv8 by package breakdown.

8

Conclusion

In this thesis, we have designed and implemented a peer to peer communication library for Android allowing to establish pure smartphone-based overlay network without any infrastructure. The networking library can establish communication with other peers either over Wi-Fi, LTE, or Bluetooth Low Energy. It builds on top of ongoing research at TU Delft, and presents the first feature-complete TrustChain implementation running natively on Android smartphones. Both the networking library and the TrustChain implementation is also fully compatible with the existing deployed network, which has been tested by running several experiments in the existing live network operated by real users. We have conducted various experiments evaluating the success rate of implemented NAT traversal, social graph scalability, bootstrap performance, and binary transfer functionality.

Furthermore, we have designed and implemented the first prototype of a distributed messaging and social networking app which demonstrates feasibility of using the library to build full-fledged alternatives to some of the most widely used services today. The app is built in the novel super app paradigm, which makes it easy to extend its functionality with mini-apps taking advantage of the existing social graph.

The project has been published as an open source with extensive code coverage and CI pipeline, allowing future contributions from the community. In the final phase of the development, we conducted an experiment by letting 4 teams of MSc students with 17 members in total develop new mini-apps taking advantage of blockchain functionality. All teams managed to develop and test functional non-trivial apps within 10 weeks, proving the ease of library use. Developed mini-apps include: decentralized autonomous organization (DAO) creation and shared ownership of money, secure online quorum-based voting, and decentralized machine learning.

The work done as part of this thesis enables a new field of research related to P2P communication on mobile devices. There is possible future work both in the context of the networking library and decentralized application development. For the networking library to be future proof, it will need to add IPv6 support, which is finally being slowly rolled out by ISPs. The networking library should also have a *Distributed Hash Table (DHT)* integrated to allow building more complex and scalable services. DHT should be in the first place used for resolving peer public keys to their IP addresses, which is needed especially in the context of mobile clients where IP addresses can change frequently. There already is a deployed implementation of a NAT-traversal-capable DHT present in py-ipv8, so porting it to Kotlin should be straightforward.

Another existing field of research is related to mesh networks. As Bluetooth only allows up to 7 connected devices, for higher number of devices, a mesh network with some form of

routing is required. Taking a step ahead, there could be even a protocol which would allow routing traffic between the mesh network and the Internet, allowing offline users to connect to peers connected over the Internet. This brings a lot of issues and would probably need to take advantage of bandwidth accounting to prevent freeriding and abuse of the protocol.

On the application level, our early versions of developed applications can be extended with more social features, allowing users to build encrypted group chats, establish friendships, and share posts that would only be readable by trusted friends.

Finally, there are other types of services that could be implemented as mini-apps in the super app, or as standalone distributed apps. One of the ideas that has already grown into a separate ongoing thesis work includes a music streaming service which cuts the middlemen out of the artist revenue distribution. The music sharing app aims to remove all proprietary infrastructure between artists and ears of listeners with a cost-free ecosystem. Another grand idea is creating a universal distributed market which would allow to trade any real-world currency for any cryptocurrency. Such an exchange should build on top of the existing trust graph to favor trades with people who have proven to be trustworthy. Equipped with the universal trust graph, we could eventually deploy decentralized alternatives to any services interacting with real-world, such as ride-sharing, apartment rental, or marketplaces of any kind. TU Delft has a long track of research related to establishing trust among strangers. It will be interesting to see where the research is heading in the next decade, and if the proposed technology stack can form the basis for disrupting Big Tech.

Bibliography

[1] Bluetooth SIG. Bluetooth core specification version 5.1. Accessed: Oct. 26, 2019, January 2019. URL https://www.bluetooth.com/specifications/bluetooth-core-specification/.

[2] Briar Project. A quick overview of the protocol stack. Accessed: Oct. 26, 2019. URL https://code.briarproject.org/briar/briar/wikis/A-Quick-Overview-of-the-Protocol-Stack.

[3] Ed. C. Perkins. Ip mobility support for ipv4, revised. Technical report, 2010. URL https://tools.ietf.org/html/rfc5944.

[4] B. Carpenter. Middleboxes: Taxonomy and issues, 2002. URL https://tools.ietf.org/html/rfc3234.

[5] Mike Cleron. Android announces support for kotlin. Accessed: June 15, 2020, 2017. URL https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html.

[6] Daxue Consulting. Payment methods in china: How china became a mobile-first nation. Accessed: July 7, 2020, 2020. URL https://daxueconsulting.com/payment-methods-in-china/.

[7] Ed. D. Wing, S. Cheshire, M. Boucadair, R. Penno, and P. Selkirk. Port Control Protocol (PCP). Technical report, 2013. URL https://tools.ietf.org/html/rfc6887.

[8] Martijn de Vos, Can Umut Ileri, and Johan Pouwelse. Xchange: A blockchain-based mechanism for generic asset trading in resource-constrained environments, 2020.

[9] R. I. M. Dunbar. Do online social media cut through the constraints that limit the size of offline social networks? 2016. URL https://royalsocietypublishing.org/doi/full/10.1098/rsos.150292.

[10] K. Egevang and P. Francis. The IP Network Address Translator (NAT). Technical report, 1994. URL https://tools.ietf.org/html/rfc1631.

[11] Ed. F. Audet and C. Jennings. Network address translation (nat) behavioral requirements for unicast udp. Technical report, 2007. URL https://tools.ietf.org/html/rfc4787.

[12] Gertjan Halkes and Johan Pouwelse. UDP NAT and firewall puncturing in the wild. In *NETWORKING 2011*, pages 1–12, Berlin, Heidelberg, 2011. ISBN 978-3-642-20798-3.

[13] Armijn Hemel. Universal plug and play: Dead simple or simply deadly? 2006. URL http://www.sane.nl/sane2006/program/final-papers/R6.pdf.

[14] Paul Huang. More evidence wechat is recording private messages for beijing to spy on users. Accessed: July 7, 2020, 2018. URL https://www.theepochtimes.com/more-evidence-wechat-is-recording-private-messages-for-beijing-to-spy-on-users_2510577.html.

[15] Dara Khosrowshahi. An operating system of everyday life. Accessed: July 7, 2020, 2019. URL https://www.uber.com/newsroom/everyday-life-os/.

[16] Protocol Labs. libp2p. Accessed: August 24, 2020. URL https://libp2p.io/.

[17] Rita Liao. Wechat reaches 1m mini programs, half the size of apple's app store. Accessed: July 7, 2020, 2020. URL https://techcrunch.com/2018/11/07/wechat-mini-apps-200-million-users/.

[18] D. MacDonald and B. Lowekamp. NAT behavior discovery using Session Traversal Utilities for NAT (STUN). RFC 5780, May 2010. URL https://tools.ietf.org/html/rfc5780.

[19] Fiona M. Scott Morton and David C. Dinielli. Roadmap for an antitrust case against facebook. Accessed: July 6, 2020, 2020. URL https://www.omidyar.com/sites/default/files/Roadmap%20for%20an%20Antitrust%20Case%20Against%20Facebook.pdf.

[20] Mitchell Olsthoorn. FBase: Trustworthy code module execution, 2020. URL https://repository.tudelft.nl/islandora/object/uuid%3Ad68197ec-50b9-4452-b9bf-e34a743f165f.

[21] Pim Otte, Martijn de Vos, and J.A. Pouwelse. Trustchain: A sybil-resistant scalable blockchain. *Future Generation Computer Systems*, 09 2017. doi: 10.1016/j.future.2017.08.048.

[22] J. Pouwelse, Pawel Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. Epema, M. Reinders, M. V. Steen, and H. Sips. Tribler: A social-based peer-to-peer system. *Concurr. Comput. Pract. Exp.*, 20:127–138, 2006.

[23] J.A. Pouwelse, J. Yang, M. Meulpolder, D.H.J. Epema, and H.J. Sips. Buddycast: an operational peer-to-peer epidemic protocol stack. In GJM Smit, DHJ Epema, and MS Lew, editors, *Proc. of the 14th Annual Conf. of the Advanced School for Computing and Imaging*, pages 200–205. ASCI, 2008. ISBN 978-90-810849-3-2. ASCI-2008-buddycast.

[24] The Briar Project. Briar – secure messaging, anywhere. Accessed: August 24, 2020. URL https://briarproject.org/.

[25] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. Stun - simple traversal of user datagram protocol (udp) through network address translators (nats). Technical report, 2003. URL https://tools.ietf.org/html/rfc3489.

[26] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session traversal utilities for nat (stun). Technical report, 2008. URL https://tools.ietf.org/html/rfc5389.

[27] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984. ISSN 0734-2071. doi: 10.1145/357401.357402. URL https://doi-org.tudelft.idm.oclc.org/10.1145/357401.357402.

[28] Statista. Distribution of mobile network connections in the netherlands in the fourth quarter of 2018, by operator. Accessed: Mar. 11, 2020. URL https://www.statista.com/statistics/765491/distribution-mobile-network-connections-in-the-netherlands-by-operator/.

[29] Y. Takeda. Symmetric NAT traversal using STUN. Technical report, June 2003. URL https://tools.ietf.org/id/draft-takeda-symmetric-nat-traversal-00.txt.

[30] Dominic Tarr. Designing a secret handshake: Authenticated key exchange as a capability system. 2015.

[31] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. Secure scuttlebutt: An identity-centric protocol for subjective and decentralized applications. In *Proceedings of the 6th ACM Conference on Information-Centric Networking*, 2019.

[32] The Fortnite Team. #freefortnite. Accessed: August 24, 2020, 2020. URL https://www.epicgames.com/fortnite/en-US/news/freefortnite.

[33] Financial Times. The economics of big tech. Accessed: July 6, 2020, 2020. URL https://www.ft.com/economics-of-big-tech.

[34] Kevin Townsend, Carles Cufí, Akiba, and Robert Davidson. *Getting Started with Bluetooth Low Energy*. O'Reilly Media, Inc., 2014.

[35] Tribler. Tribler/py-ipv8: Python implementation of the ipv8 layer. Accessed: June 13, 2020. URL https://github.com/Tribler/py-ipv8.

[36] DONALD J. TRUMP. Executive order on addressing the threat posed by wechat. Accessed: August 24, 2020, 2020. URL https://www.whitehouse.gov/presidential-actions/executive-order-addressing-threat-posed-wechat/.

[37] W3C. ActivityPub. Accessed: August 24, 2020, 2018.

[38] Wi-Fi Alliance. Wi-Fi Peer-to-Peer (P2P) technical specification, version 1.7. June 2016. URL https://www.wi-fi.org/file/wi-fi-peer-to-peer-p2p-technical-specification-v17.

[39] Wi-Fi Alliance. Neighbor Awareness Networking specification, version 3.0. December 2018. URL https://www.wi-fi.org/file/neighbor-awareness-networking-specification.

[40] Niels Zeilemaker, Boudewijn Schoon, and Johan A. Pouwelse. Dispersy bundle synchronization. 2013.