Ubiquitous Overlay

Universal communication using imperfect hardware

Matouš Skála

TU Delft

Ubiquitous Overlay

Universal communication using imperfect hardware

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
	?,	TU Delft
	?,	TU Delft

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

TU Delft

Preface

Preface...

Matouš Skála
Delft, April 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 power to take the network down. Yet, 50 years later, we live in the 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 the 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 communication between any two devices. It is implemented as a Kotlin library which can be used on 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 NATs. When the Internet connection is not available and peers are located in proximity, the connection can be established using Bluetooth Low Energy. Peers are addressed by their public keys and their physical addresses at lower layers are abstracted away.

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 capable of establishing a connection in all tested network conditions.

To demonstrate the usage of the library, a decentralized messaging application with end-to-end encryption, which allows trustworthy communication insusceptible to the man-in-the-middle attack, 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

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 ability to turn it off. The Internet standards are developed by the working groups of Internet Engineering Task Force (IETF), a non-profit open standards organization composed of volunteers. Its evolution is based on a rough consensus about technical proposals, and on running code. While there are many conflicting opinions on its architecture, the general consensus is that the main goal of the Internet is to provide global connectivity by the *Internet Protocol (IP)*. [3] This ultimate goal requires cooperation of multiple parties, including researchers, developers, and commercial service providers.

2.1. End-to-End Principle Challenged

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." [17]

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. A common example for supporting the end-to-end principle are delivery guarantees. In the ARPANET, the receipient would send *Request For Next Message* packet every time a packet was delivered. However, some applications would still need to implement its own acknowledgements on the application level, to indicate that the messages were processed correctly. Similar examples can be found for supporting authentication, encryption, and other guarantees in communication systems.

The end-to-end principle has been challenged by 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 gateways, 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 severly 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.

2.2. Overcoming Address Exhaustion

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 IPv4 address exhaustion, internet providers were forced to deploy different types of NATs which allows 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 ports that are explicitly open, 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 internet service providers to implement a NAT in their infrastructure. This topology is called a *carrier-grade NAT* (CGN). Carrier-grade NAT is by definition managed by the network operator and the customer does not have any control over it. CGN usually implement a complex functionality to ensure reliability for thousands 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)* [6] 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 [8], which makes P2P communication nearly impossible.

2.3. 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 of them more worrying than others. There are several countries that enforce Internet censorship either by simple IP address blocking, or advanced deep packet inspection, preventing people from communicating freely.

Even in places with relatively mature infrastructure, there are occasionally connectivity issues at places with high gatherings 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. 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 adopt to a peer-to-peer system with self-sovereign identities.

2.4. Re-decentralization of Internet Infrastructure

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. The majority of the websites is operated by a few hosting companies and most of the static content is served by a few *Content Delivery Networks (CDNs)*. For example, *Cloudflare* currently powers 10 % of all internet traffic[1]. Since there is a central point of failure, if a provider experiences outage, a large part of the internet goes down.

The inventor of the World Wide Web has proposed a Solid [13] as an architectural pattern for building the next generation of decentralized social applications. It is based on the principle that users own their data and have freedom of choice where to store them, which significantly improves privacy and data reusability. As the application logic is decoupled from the data storage servers, multiple applications can access them same data if allowed by the user.

2.5. Research Question

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

- **Can we devise a protocol facilitating trustworthy device-to-device communication between smartphones under challenging network conditions without using a central server?**

We analyze the question in more detail:

- The communication should be **trustworthy**, which means we can verify that we received the message from the person that claims to be its sender. The communication can be optionally encrypted to provide privacy on top of authenticity.
- The communication should be facilitated **device-to-device** without need for any additional **server** infrastructure, with the exception of the initial bootstrap server. The bootstrap server should not be needed if there are peers in proximity which can be used for bootstrapping.
- The protocol should be able to work under **challenging network conditions**, which means it should be robust against the presence of various middleboxes. Mainly, it should be able to establish connection in presence of carrier-grade NATs deployed by mobile network operators.
- The communication protocol should in theory work on both most popular **smartphone** operating systems and support their most commonly used versions. However, supporting only the most commonly used OS would be sufficient for the prototype.

[1] https://blog.cloudflare.com/cloudflare-traffic/

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 problem of network address translation (NAT) and 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

There are many projects dealing with the problem of peer to peer (P2P) communication on different levels, ranging from general-purpose libraries that can be used to build applications (libp2p, IPv8), to feature-packed platforms providing social networking features on top of their custom-designed protocols (Briar, Bridgefy, Berty, Secure Scuttlebutt). We try to analyze their functionality from the technical and usability perspective, state their advantages and shortcomings, and propose improvements when possible.

3.1.1. Fediverse

TODO: federated servers used for web publishing (Mastodon, PeerTube, Friendica)

3.1.2. libp2p

libp2p[1] is a modularized peer-to-peer networking stack and library developed by Protocol Labs, a non-profit company behind the Interplanetary File System (IPFS)[2], a distributed filesystem comibing 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. This 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 both on JVM and Android runtime.

The library is composed of several layers, where each can be implemented by one of interchangeable modules. 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

[1] https://libp2p.io
[2] https://ipfs.io

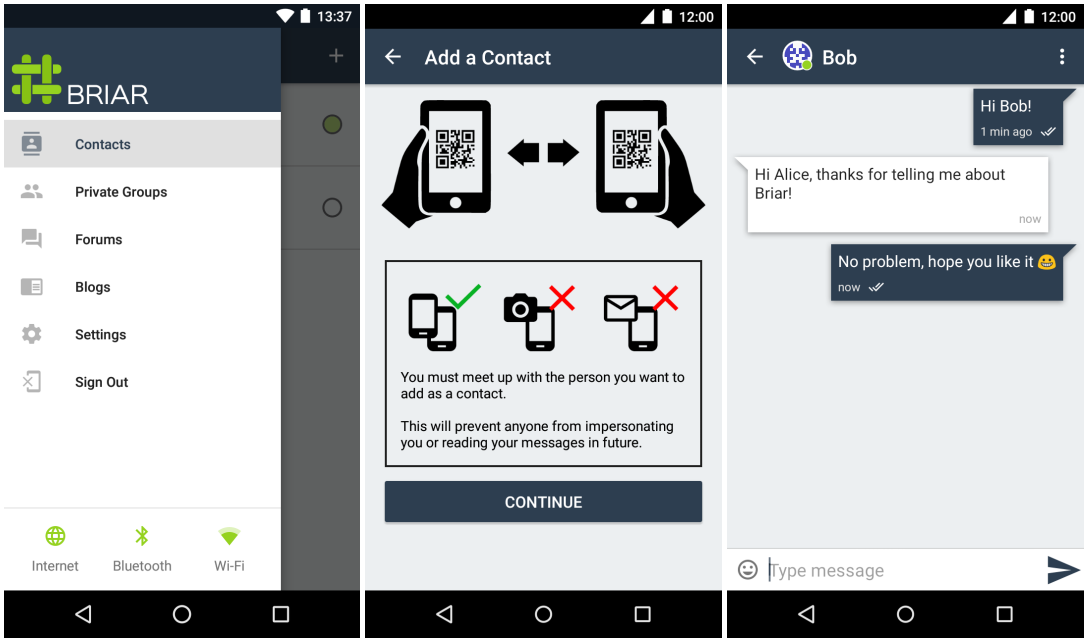

Figure 3.1: The menu, contact exchange, and the conversation detail in the Briar app

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.

3.1.3. Briar

Briar [**?**] 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 only accepts connections from devices in contacts. However, the user can 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*. [**?**] 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.

3.1.4. Secure Scuttlebutt

Scure Scuttlebutt (SSB) [21] is a peer-to-peer gossip protocol for building decentralized applications. Its first application was a distributed social networking platform Scuttlebutt. 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.

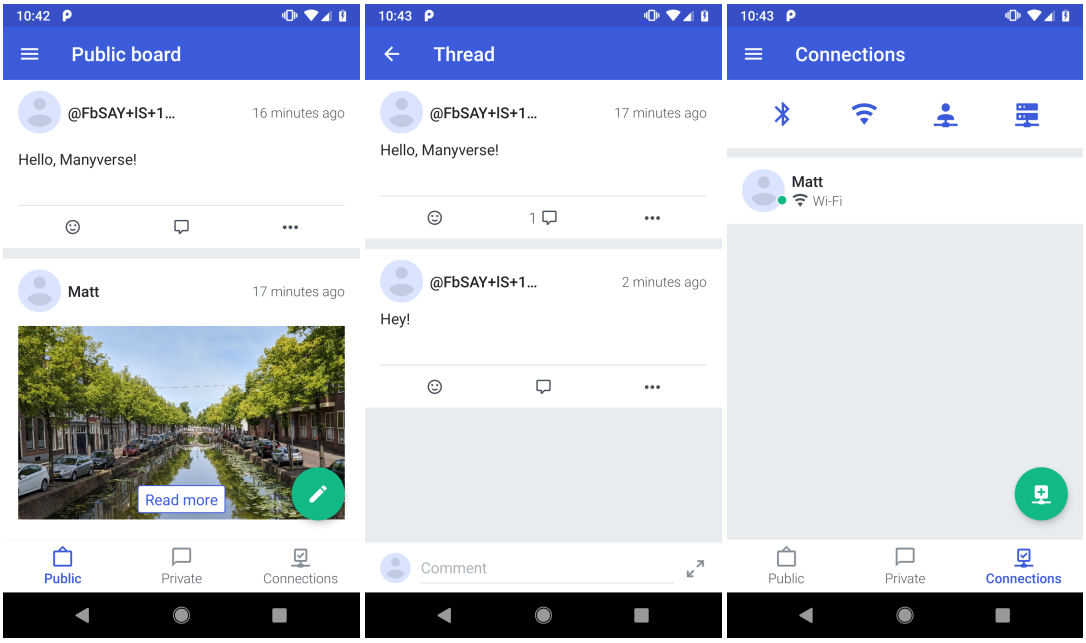

Figure 3.2: The feed, thread, and connections list UI in the Manyverse Android app

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.

Peer Discovery

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 [20]. Subsequently, they can communicate using an RPC protocol. The peer can then access feeds of other peers followed by the same pub.

Blobs

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.

Private Messages

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 one of the intended recipients.

Conclusion

The reference protocol implementation is written in Node.js, with ongoing efforts for Go, Python, and Rust implementations. There is an official multiplatform UI client Patchwork[3] implemented in the Electron framework. There is also a mobile client called Manyverse[4] written in React Native, available both for Android and iOS.

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. 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.

3.1.5. IPv8

IPv8 [23] is a P2P networking library developed over the last 13 years at TU Delft. It is an evolution of previous generations of *BuddyCast* [12] and *Dispersy* [25]. Its primary purpose is to serve as the networking layer of Tribler, 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:

Identity

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.

NAT Traversal

UDP hole punching is implemented as a basic NAT traversal technique to allow connectivity between peers behind NATs.

Peer Discovery

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.

TrustChain

The library also provides TrustChain [**?**], 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 [**?**], or identity attestations and verifiable claims.

Mobile Support

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

[3] https://scuttlebutt.nz/
[4] https://www.manyver.se/

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 uses the same principles.

3.2. Network Address Translation

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. [7] Yet, it has become so ubiquitous in the Internet infrastructure that almost all end users are located behind a NAT nowadays.

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 estensively 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.

3.2.1. NAT Classification

The original 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 [8].

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.

[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

- **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 [8] 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. [9] 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 [8] 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.

3.3. NAT Traversal

THe 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.

3.3.1. 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. [10]

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 superceeded by the *Port Control Protocol (PCP)*, which was standardized in 2013. [6] 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.

3.3.2. Session Traversal Utilities for NAT (STUN)

STUN was originally defined in [15] as *Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. It was a client–server protocol that allowed ap-

plications 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 [16]. 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.

3.3.3. 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 screnarios, 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.

3.3.4. Interactive Connectivity Establishment (ICE)

The *Interactive Connectivity Establishment (ICE)* [14] provides the complete solution for NAT traversal which is built on top of STUN and TURN protocols. TODO

3.3.5. Symmetric NAT Traversal

3.4. Nearby 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 3.3.

3.4.1. 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

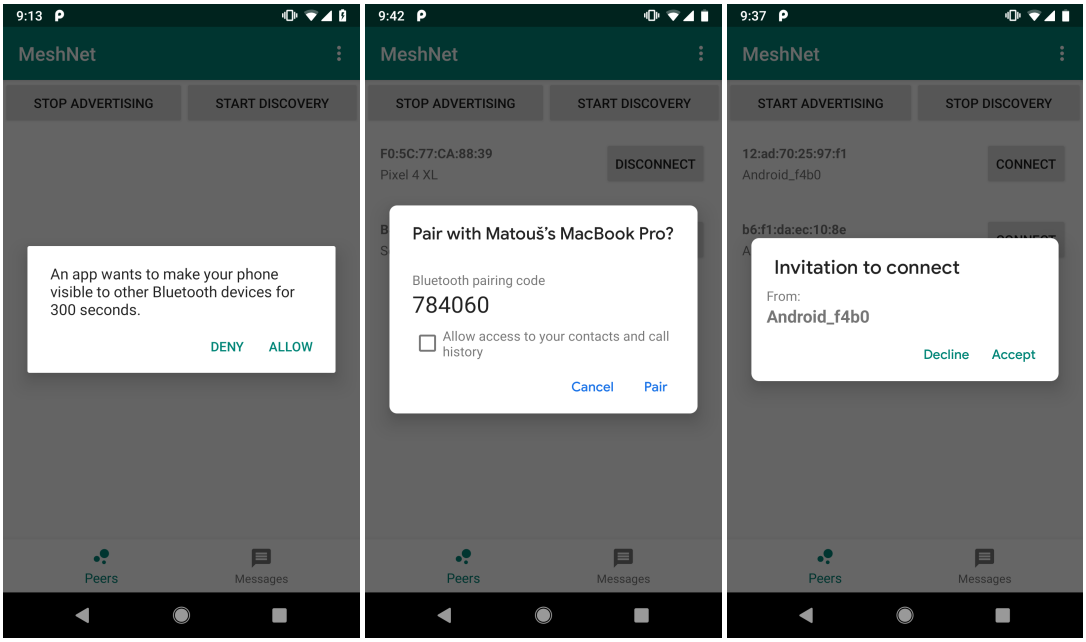

Figure 3.3: The system UI for enabling Bluetooth discovery, accepting an incoming Bluetooth pairing request, and accepting an incoming Wi-Fi Direct connection on Android 9

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.

3.4.2. Bluetooth Low Energy

Bluetooth Low Energy (BLE) [22] 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.

3.4.3. 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.

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

Wi-Fi Direct (also known as Wi-Fi Peer-to-Peer) [**?**] 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.

3.4.4. Wi-Fi Aware

Wi-Fi Aware, also known as *Neighbor Awareness Networking* (NAN) is a recent networking standard introduced by Wi-Fi Alliance. [**?**] 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

In this chapter, we design a protocol facilitating peer to peer communication between any devices. 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. In the end, we introduce the architecture of the whole system and describe the interaction between all individual building blocks.

It is important to note that in this thesis, we extend an existing IPv8 protocol [23]. 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* [2] 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 acts 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 finiding 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.

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 until they 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 called *datagrams* and its 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. UDP can also be used for building higher-level transport protocols such as *uTP* or *QUIC*.

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. ICE 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

4.3.3. Authentication and Encryption

4.3.4. Binary File Transfer over UDP

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 [9] and [25]. 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.

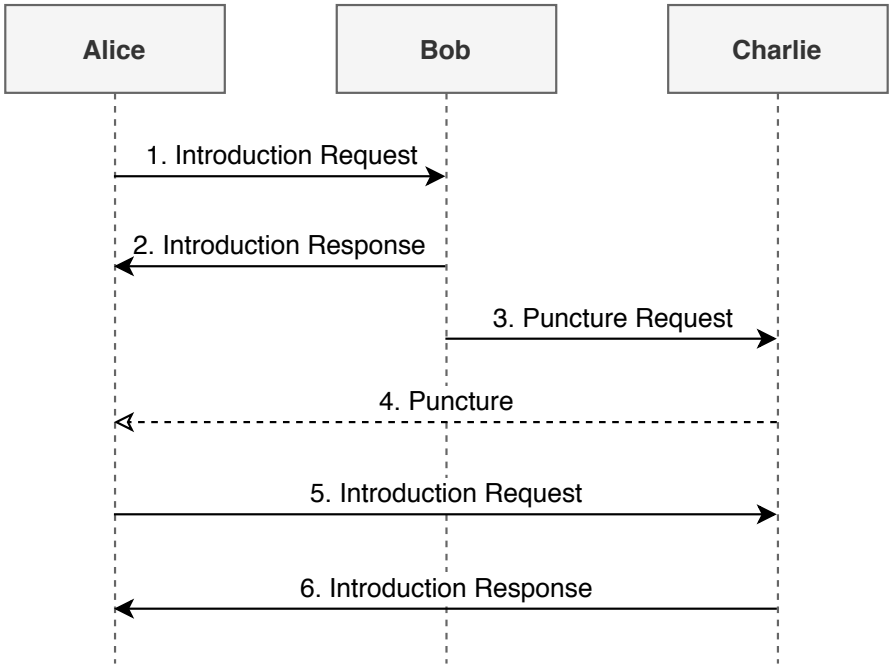

Figure 4.2: Peer discovery mechanism with NAT traversal

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

Inspired by the previously discussed symmetric NAT traversal methods, 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 the 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.

In this case, we 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.

Even in the extreme case when the NAT assigns ports completely at random, our NAT traversal mechanism should still work.

The problem is that when a peer A is connecting to peer B behind a symmetric NAT, the peer A does not have a way to find out the public port of peer B.

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. 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.

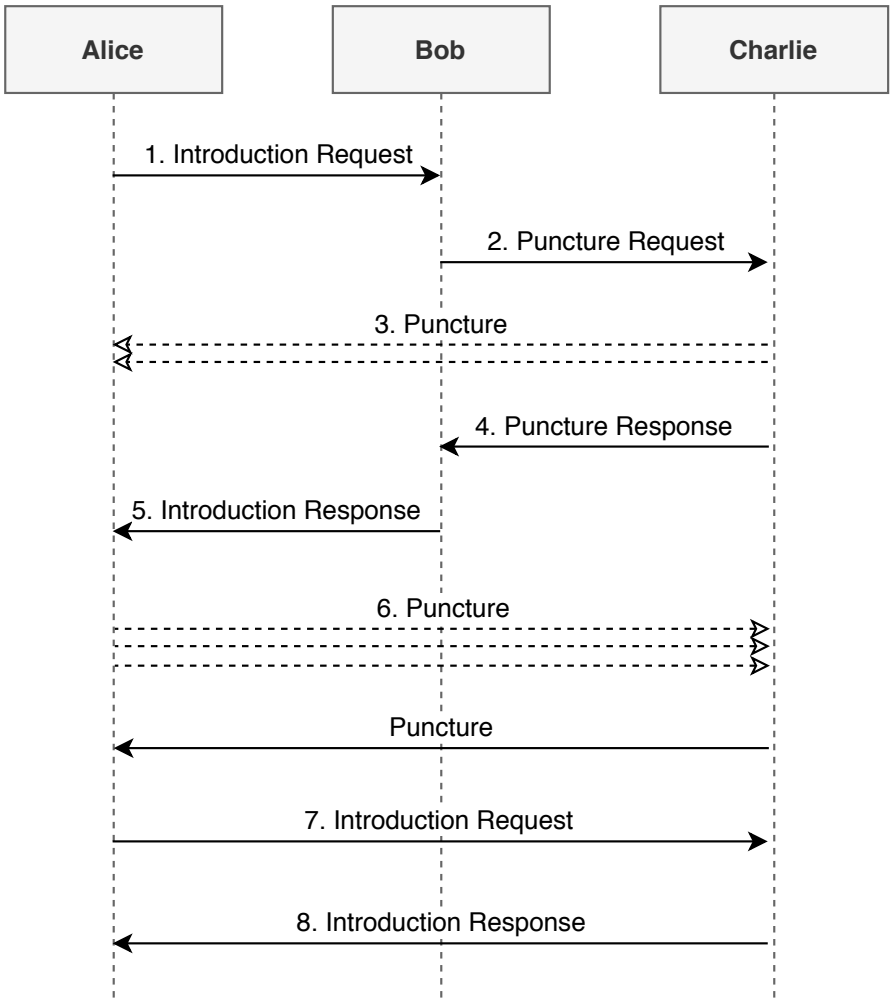

Figure 4.3: The flow diagram of the symmetric NAT traversal protocol

4.5.3. Extended Peer Introduction Protocol

4.6. Security Considerations

4.6.1. Sybil Attack

4.6.2. Eclipse Attack

4.6.3. Distributed Denial of Service

4.6.4. WAN Address Forgery

4.7. P2P Communication with Nearby Devices

4.7.1. 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 repatedly 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.

4.7.2. BLE Protocol Stack

The BLE standard is composed of several protocols which form the BLE stack visualized in Figure 4.4. The stack is split in two parts: a *controller* and a *host*. The controller is usually implemented in hardware, while the host is a part of the operating system. Both parts communicate over the *Host Controller Interface (HCI)*. Even though the applications usually communicate only with the highest layers, it is worth to understand how all layers of the stack fit together. In this section, we describe the responsibilities and mechanics of each protocol.

Logical Link Control and Adaptation Protocol (L2CAP)

On the lowest layer in the host, L2CAP is responsible for multiplexing and splitting data from higher-level protocols (ATT and GAP) into 27-byte data packets. These are then forwarded to the Link Layer and transmitted over the Physical Layer implemented in the Bluetooth radio. Starting with Bluetooth 4.1, the applications can communicate over L2CAP directly. User-defines channels allow for higher-throughput data transfer without the additional complexity added by ATT.

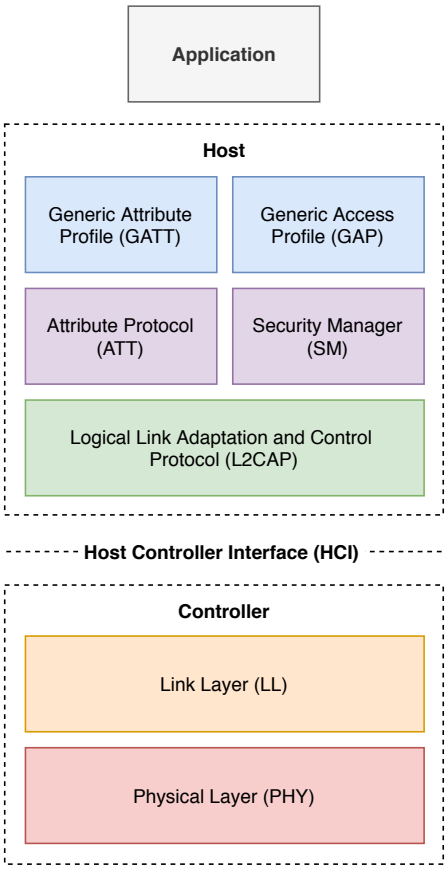

Figure 4.4: Bluetooth Low Energy Protocol Stack [22]

Attribute Protocol (ATT)
ATT is a client-server protocol for reading and writing attributes. It is strict about sequential operation. If a request is pending, no further requests should be sent until the response is received, otherwise they are discarded. Attributes are stored under attribute handles defined by UUIDs. Various operations are defined by the protocol. Apart from the standard configuraiton, read and write, queued write is supporte to write data longer than a single packet. If a client wants to be notified about handle changes, it can subscribe to handle value notifications or indications. Both allow the server to notify the clients whenever a value changes, but in case of indication, the client should respond with ack packet to confirm it has received the indication.

Generic Attribute Profile (GATT)
The Generic Attribute Profile defines how to exchange data between devices. It uses ATT as a transport protocol. The data are organized hierarchically in *services* which contain groups of related data called *characteristics*, which can be further specified using *descriptors*.

GATT is a client–server protocol and it follows the same principles as ATT. A client sends requests to the server and receives responses or server-initiated updates. The server is responsible for storing data written by the client and responding to read requests. It is important to mention that the client and server roles in GATT are independent of the roles defined by GAP. Both a peripheral and a central can take the role of a client or a server, or even both at the same time.

4.7.3. 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 7.1.

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

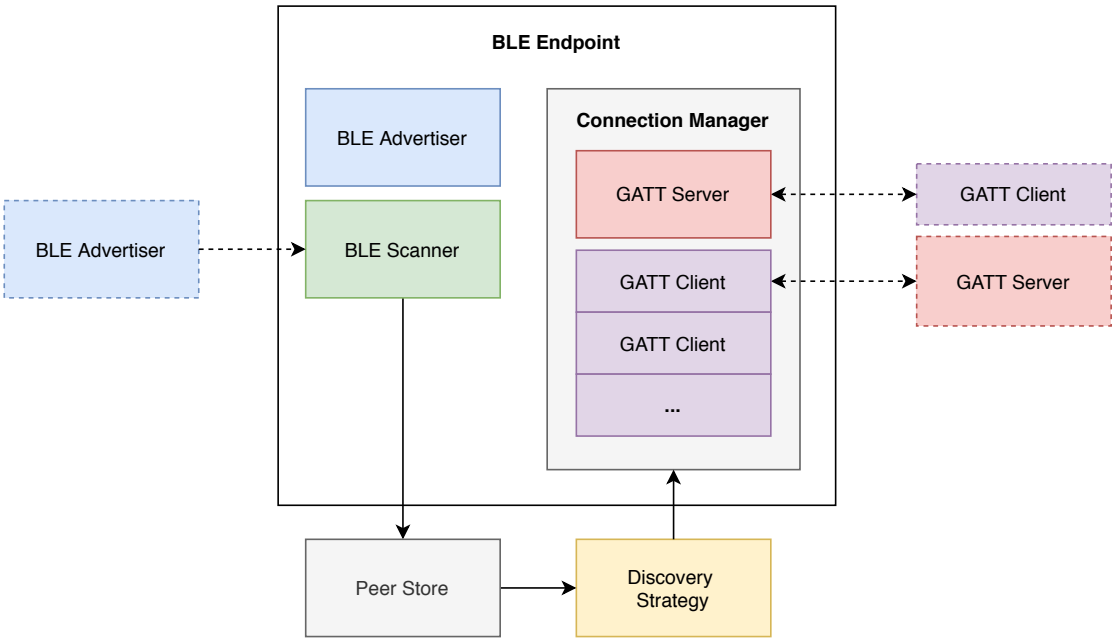

Figure 4.5: Bluetooth Low Energy Communication Architecture

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 device it is connected to. The GATT server implements a single GATT service containing two characteristics which are shown in Figure 4.6. 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.

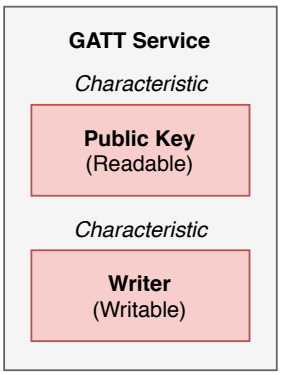

Figure 4.6: Our GATT Server Architecture

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. 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-

[1] https://www.rust-lang.org/
[2] https://reactnative.dev/
[3] https://kotlinlang.org/
[4] https://github.com/Tribler/kotlin-ipv8

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. The ipv8 module has extensive code coverage of 72%.

5.2. 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.

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 used to perform the communication. 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] https://junit.org/
[6] https://mockk.io/
[7] https://ktlint.github.io
[8] https://codecov.io

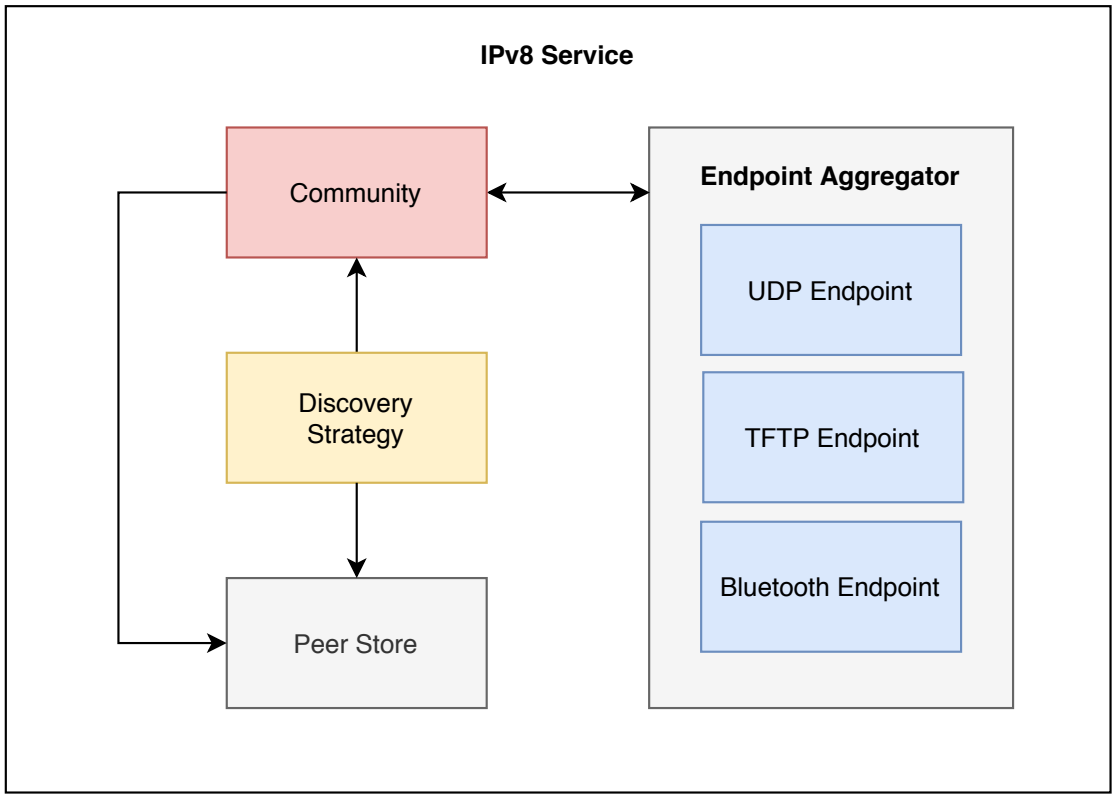

Figure 5.1: The system architecture of the IPv8 service

5.3. Communities

5.3.1. Community

Each community 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.
- `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.

5.3.2. 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.

5.4. Discovery Strategies

5.4.1. 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.

5.4.2. 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 for `DiscoveryCommunity`.

5.4.3. 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. [9]

5.4.4. Bluetooth Discovery Strategy

5.5. Endpoints

5.5.1. Endpoint Aggregator

5.5.2. UDP Endpoint

5.5.3. TFTP Endpoint

5.5.4. Bluetooth Endpoint

5.6. 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
```

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

5.7. Library Usage

In this section, we describe the usage of the kotlin-ipv8 library from the perspective of the application developer. We will show how to create a simple overlay network, and send and handle custom messages. Finally, we will configure and start the IPv8 stack, and load our overlay with a discovery strategy, which will allow us to discover other peers in the community using the bootstrap server.

5.7.1. Project Setup

To include the library in an Android app, the application developer should first include the kotlin-ipv8 repository as a Git submodule. In the future, the library could be deployed to a maven repository. At that point Git submodule would not be needed and the application could depend on a specific version of the library.

The Android application module should depend on the ipv8-android module which includes and exposes APIs of the ipv8 module, and defines some Android-specific dependencies and helper classes. The dependency can be defined by adding the following line to `build.gradle`:

```
implementation project(':ipv8-android')
```

5.7.2. Creating a Community

All communities have to extend an abstract `Community` class, which implements the `Overlay` interface. The only field left for us to define is `serviceId`. This can be an arbitrary 20-byte array represented as a hexadecimal string that uniquely identifies the community. We therefore generate a serviceId and define a `DemoCommunity` class:

```
class DemoCommunity : Community() {
    override val serviceId = "02313685c1912a141279f8248fc8db5899c5df5a"
}
```

5.7.3. Generating a Private Key

Every peer in has to generate a key pair which establishes the identity and is used to sign and verify messages. We can generate a private key with `AndroidCryptoProvider.generateKey()`. The key can then be serialized to a byte array with `Key.keyToBin()` method, and again deserialized using `AndroidCryptoProvider.keyFromPrivateBin(ByteArray)`. When the app is launched for the first time, the key should be generated and persisted. The previously generated key should be loaded on subsequent launches.

5.7.4. Library Initialization

We now proceed to initialize IPv8 and load our overlay. While it is possible to instantiate `IPv8` class directly, on Android it is easier to prepare `IPv8AndroidFactory` and call `IPv8Android.init` method to initialize the stack.

First, we define a configuration for our overlay `OverlayConfiguration` consists of an overlay factory and a list of discovery strategies. We can either define our own factory extending `Overlay.Factory` if we need to provide custom parameters to the overlay, or use the default implementation. Then, we create a factory for a discovery strategy. We use `RandomWalk`, a simple strategy discovering peers by performing a random walk in the network.

```
val demoCommunity = OverlayConfiguration(
    Overlay.Factory(DemoCommunity::class.java),
    listOf(RandomWalk.Factory())
)
```

Then, we define `IPv8Configuration`. The only required parameter is a list of overlays we want to load when the service is started. We pass the configuration created in the previous step.

```
val config = IPv8Configuration(overlays = listOf(
    demoCommunity
))
```

Finally, we create `IPv8AndroidFactory`, set the previously created configuration, our private key, and call the `init` method. This will start an IPv8 instance and create a foreground Android service to allow it to run even when the app is in the background.

```
IPv8Android.Factory(this)
    .setConfiguration(config)
    .setPrivateKey(getPrivateKey())
    .init()
```

The init method will return an instance of `IPv8Android` that can be used to access loaded overlays and communicate with them from the UI code. Alternatively, the IPv8 singleton instance can also be accessed with `IPv8Android.getInstance()`.

5.7.5. Sending and Receiving Messages

Now that we have multiple peers connected, we can perform some communication. First, we define a custom payload type implementing `Serializable` and `Deserializable` interfaces:

```
class MyMessage(val message: String) : Serializable {
    override fun serialize(): ByteArray {
        return message.toByteArray()
    }

    companion object Deserializer : Deserializable<MyMessage> {
        override fun deserialize(
            buffer: ByteArray,
            offset: Int
        ): Pair<MyMessage, Int> {
            return Pair(MyMessage(buffer.toString(Charsets.UTF_8)),
                buffer.size)
        }
    }
}
```

Next, we define a function in the DemoCommunity class which will iterate over all peers and send a message to all of them. We also define a message ID that is included as a prefix to the serialized message. The same ID will be used later to register a message handler for this message type.

```
private const val MESSAGE_ID = 1

fun broadcastGreeting() {
    for (peer in getPeers()) {
        val packet = serializePacket(MESSAGE_ID, MyMessage("Hello!"))
        send(peer.address, packet)
    }
}
```

Finally, we add a message handler to parse incoming messages and print their sender and content to the log.

```
init {
    messageHandlers[MESSAGE_ID] = ::onMessage
}

private fun onMessage(packet: Packet) {
    val (peer, payload) = packet.getAuthPayload(MyMessage.Deserializer)
    Log.d("DemoCommunity", peer.mid + ": " + payload.message)
}
```

6

Decentralized Super App

The application follows the concept of a super app, and bundles many mini-applications in a single APK. It is implemented in a separate repository *trustchain-superapp*[1] and includes kotlin-ipv8 as a Git submodule.

6.1. TrustChain: Scalable Distributed Ledger

6.1.1. TrustChain Explorer

6.2. PeerSocial: Decentralized Social Network

6.2.1. Trustworthy Friendship Establishment

6.2.2. PeerChat: Private Messaging Protocol

6.2.3. Public Post Feed

6.3. DelftDAO: Framework for Permissionless Economic Activity

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

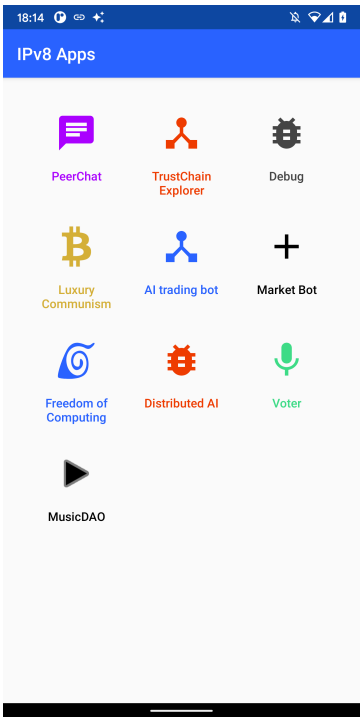

Figure 6.1: The dashboard of the super app composed of several decentralized mini-apps implemented on top of the kotlin-ipv8 library and TrustChain.

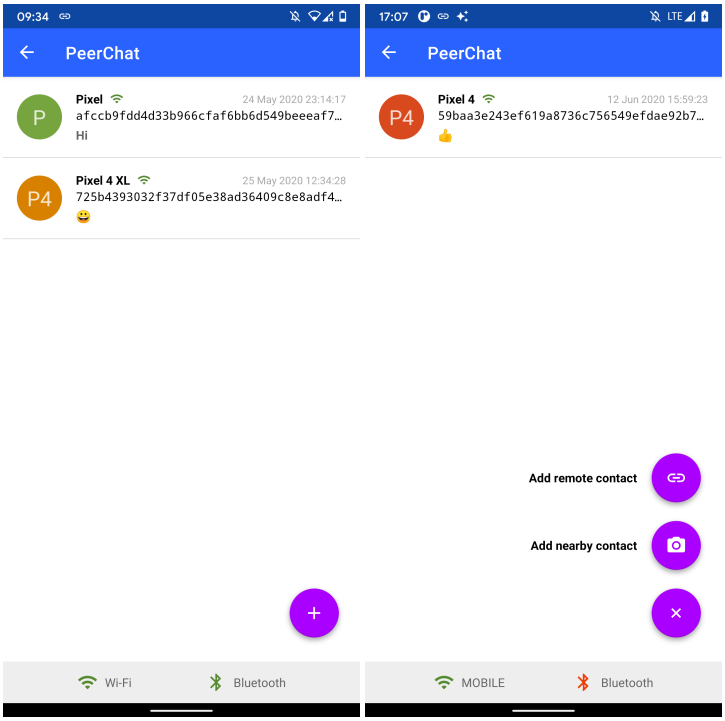

Figure 6.2: The list of contacts with online indicators, their latest messages with timestamps, and the connectivity status of the local UDP and Bluetooth endpoints.

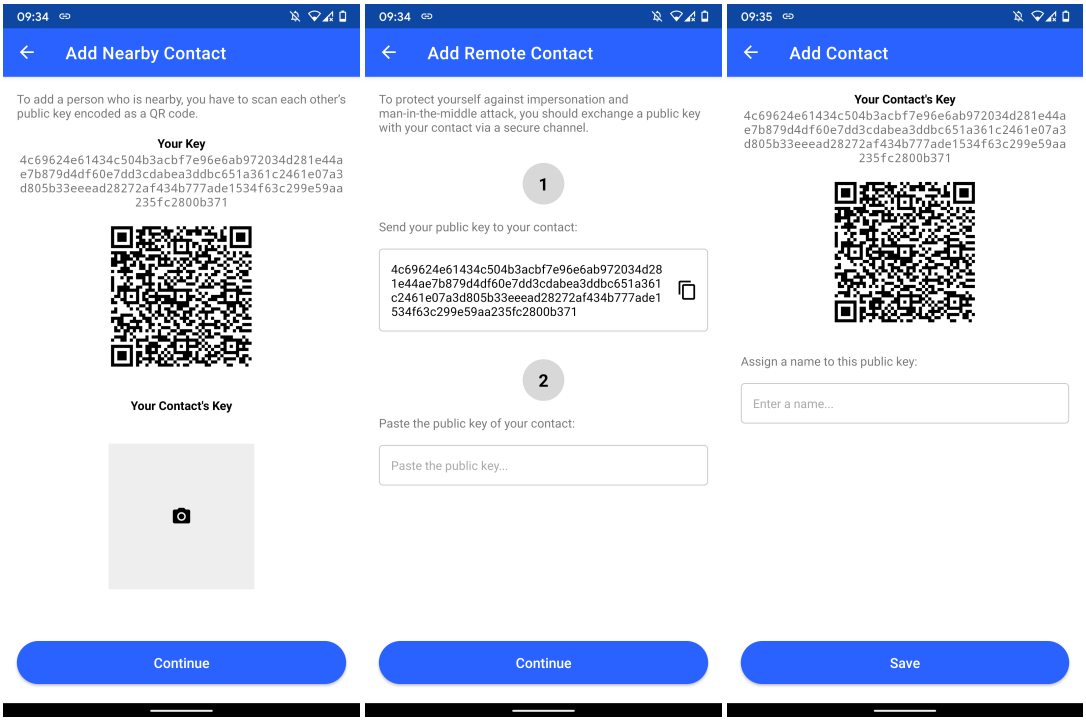

Figure 6.3: The UI for adding a public key of a new nearby or remote contact to the trusted public key store, and associating the public key with a human-readable contact name.

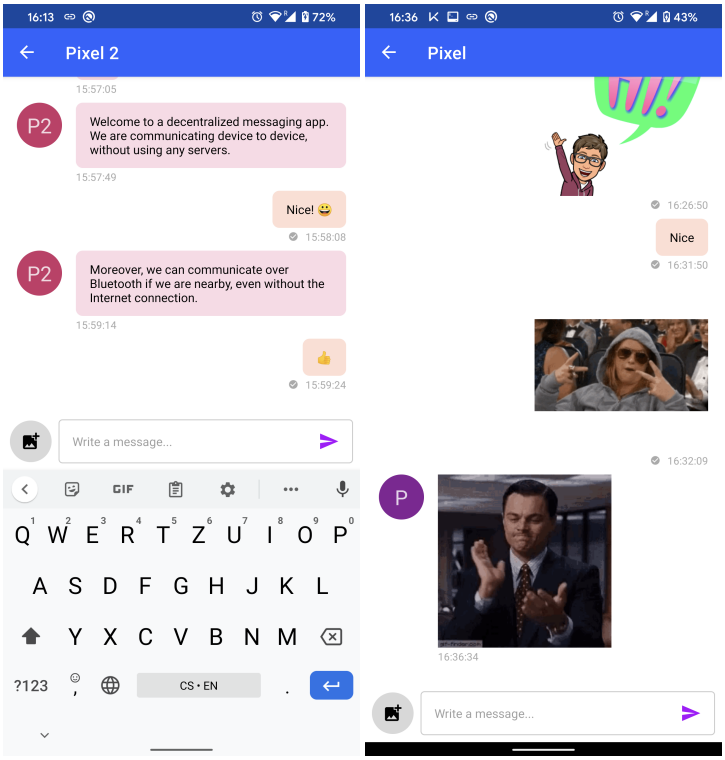

Figure 6.4: The conversation UI containing incoming and outgoing text messages, image attachments, and delivery indications for outgoing messages.

7

Evaluation

7.1. Analysis and Puncturing of Carrier Grade NAT

According to the report by Statista [18], 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 [11]. 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.

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 from the range of 10,000 ports, which makes it infeasible to use any known symmetric NAT traversal techniques such as port prediction or multiple hole punching [24] [19].

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. [18]

Operator	Mapping behavior	Filtering behavior	Binding lifetime
KPN	EIM	APDF	?
Vodafone	EIM, refresh every 60 s	APDF	?
T-Mobile	EIM	APDF	?

Table 7.2: Characteristics of CGNATs deployed by Dutch mobile network operators

7.2. Connectivity Test

7.2.1. Experimental Setup

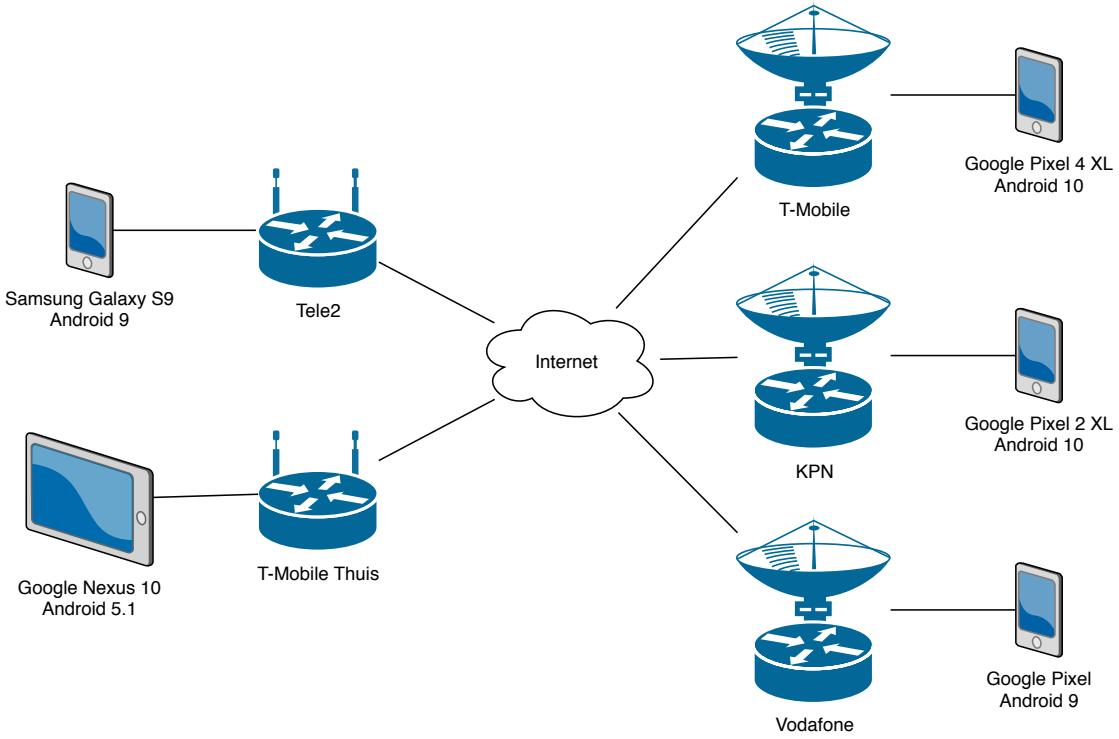

Figure 7.1: The experimental setup for the connectivity test using devices connected to different mobile (T-Mobile, KPN, Vodafone) and home broadband (Tele2, T-Mobile Thuis) ISP networks.

7.2.2. Results

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

Table 7.3: 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).

7.3. Bootstrap Performance Evaluation
7.4. Stress Test
7.5. Power Efficiency
7.6. Code Quality

8

Conclusion

8.1. Future Work

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

[3] B. Carpenter. Architectural principles of the internet. Technical report, 1996. URL https://tools.ietf.org/html/rfc1958.

[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] 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.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] The Solid Project. Solid. Accessed: May 13, 2020. URL https://solid.mit.edu/.

[14] J. Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. Technical report, 2010. URL https://tools.ietf.org/html/rfc5245.

[15] 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.

[16] 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.

[17] 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.

[18] 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/.

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

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

[21] 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.

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

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

[24] Yuan Wei, Daisuke Yamada, Suguru Yoshida, and Shigeki Goto. A new method for symmetric NAT traversal in UDP and TCP. 2008.

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