Ubiquitous Overlay

Universal connectivity using imperfect hardware

Matouš Skála

Ubiquitous Overlay

Universal connectivity 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 Tuesday January 1, 2013 at 10:00 AM.

Student number:	4893964	
Project duration:	November 15, 2019 – June 30, 2020	
Thesis committee:	Dr.ir. J.A. Pouwelse,	TU Delft, supervisor
	?,	TU Delft
	?,	TU Delft

TU Delft

Preface

Preface...

Matouš Skála
Delft, April 2020

Contents

1
Introduction

// TODO: rewrite introduction

The Internet was created with the idea that any two computers connected to the common network should be able to communicate with each other. 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 *Network Address Translation (NAT)* which allows a single address to be shared across multiple devices.

Another issue appeared with the rise of portable computers and smartphones. IPv4 addresses are dependent on the physical location and can not be considered stable user identifiers. There has been several proposals including Mobile IP, IPSec, and IPv6 to improve the usability and security of the Internet Protocol. However, none of them have been widely deployed yet or address all known issues.

This thesis proposes and implements a decentralized protocol for peer to peer communication between any two devices. It is implemented in form of a Kotlin library which can be used on desktop, smartphones, tablets, and IoT devices. This library 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 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 on lower layers are abstracted away.

The protocol makes best effort to connect peers behind NATs. However, in case the connection is not possible, it resorts to a relay protocol. Bandwidth accounting prevents misusing the relay servers and provides incentive for relay operators. The protocol is completely decentralized and does not rely on any central entity.

To show one of many practical use cases of the protocol, a simple chat messaging application is implemented on top of it. It allows two users to exchange identities in a secure way to prevent MITM attack. Then they can transfer not only text messages, but also images and videos, to demonstrate a binary file transfer.

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, and is completely open source.

Finally, the protocol performance is experimentally evaluated with multiple Android devices connected to different Wi-Fi and carrier networks and running a stress test over the period of 24 hours.

2 Problem Description

2.1. Ubiquitous Overlay Network

2.2. Network Address Translation

2.2.1. NAT Classification

2.2.2. Carrier Grade NAT

2.2.3. Port Forwarding

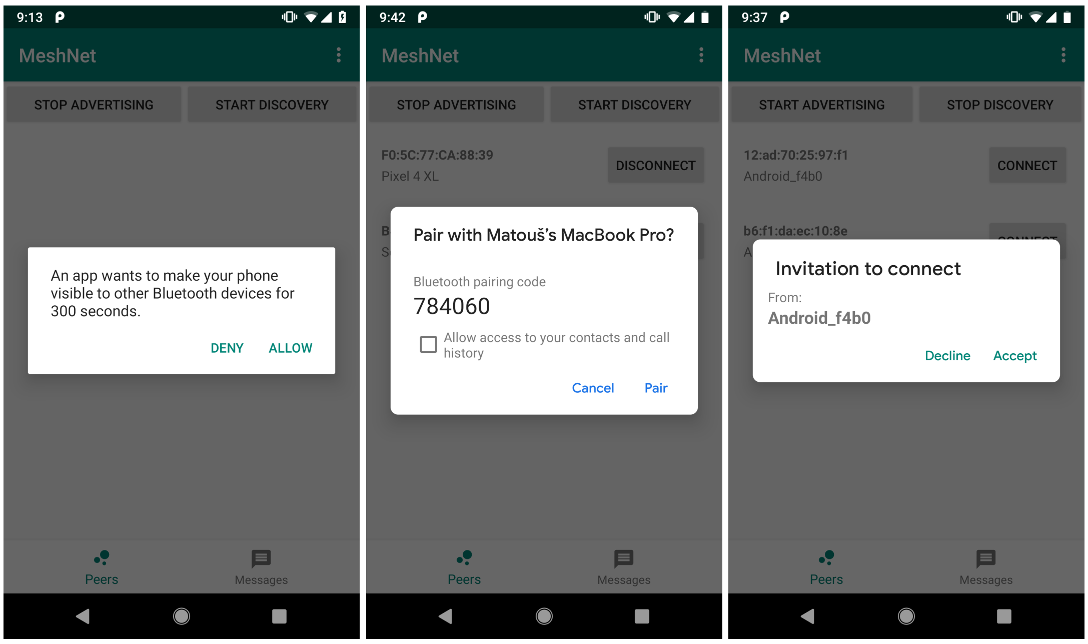
2.3. Nearby Communication

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 2.1.

2.3.1. Bluetooth

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

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



An app wants to make your phone visible to other Bluetooth devices for 300 seconds.

Pair with Matouš's MacBook Pro?

Bluetooth pairing code

784060

Allow access to your contacts and call history

Invitation to connect

From:

Android_f4b0

Figure 2.1: The system UI for enabling Bluetooth discovery, accepting an incoming Bluetooth pairing request, and accepting an incoming Wi-Fi Direct connection on Android 9

MAC address of the other device is known, the user of the other device still needs to manually set it to be discoverable.

2.3.2. Bluetooth Low Energy

Bluetooth Low Energy (BLE) [5] 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.

2.3.3. Wi-Fi Direct

2.3.4. Wi-Fi Aware

2.3.5. Support in Operating Systems

2.4. Peer Discovery

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

3
State of the Art

4 Design

4.1. NAT Traversal with Peer Introductions

4.2. P2P Communication with Nearby Devices

4.2.1. 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.

4.2.2. BLE Protocol Stack

The BLE standard is composed of several protocols which form the BLE stack visualized in Figure 4.1. 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

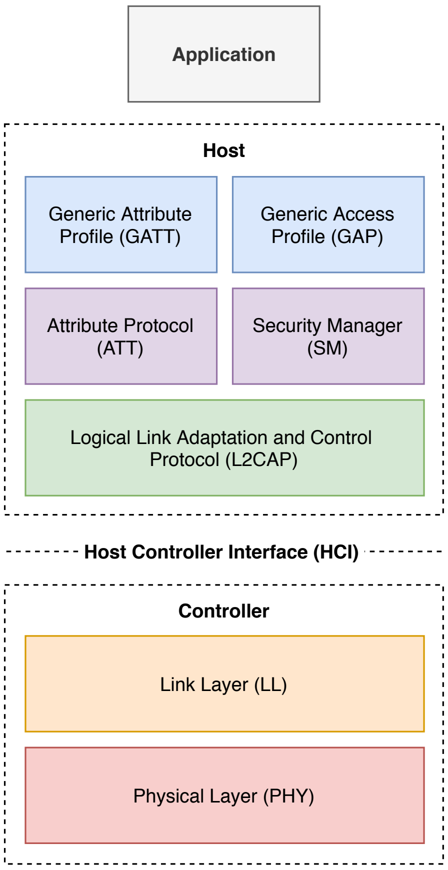

Application
Host
Generic Attribute Profile (GATT)
Generic Access Profile (GAP)
Attribute Protocol (ATT)
Security Manager (SM)
Logical Link Adaptation and Control Protocol (L2CAP)
Host Controller Interface (HCI)
Controller
Link Layer (LL)
Physical Layer (PHY)

Figure 4.1: Bluetooth Low Energy Protocol Stack [5]

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.

Generic Access Profile (GAP)
// TODO

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

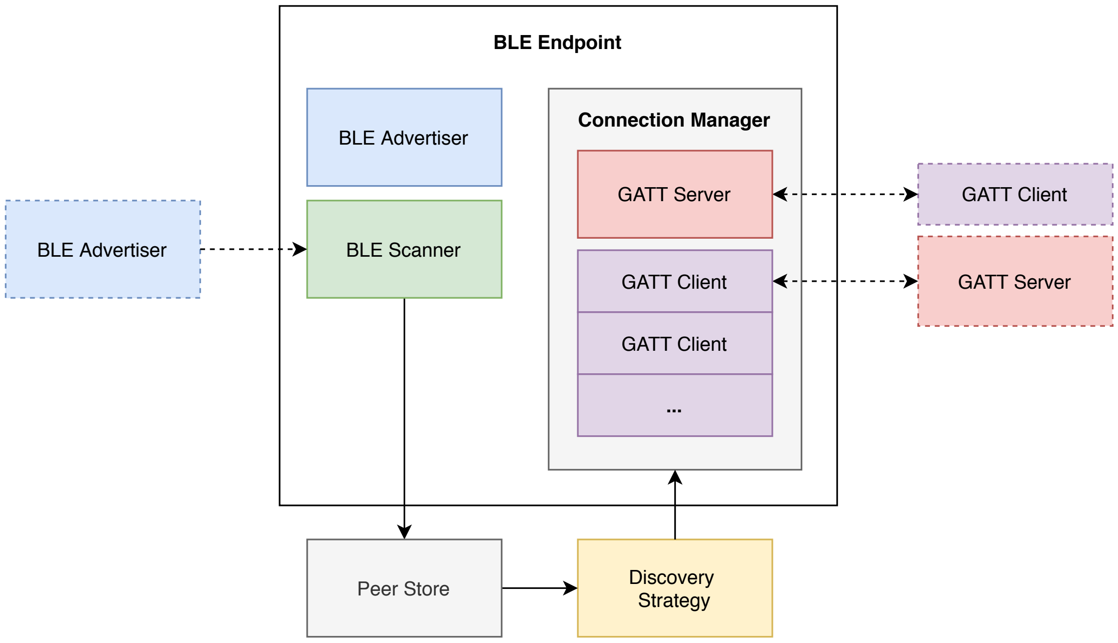


Figure 4.2: Bluetooth Low Energy Communication Architecture

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.

4.2.3. 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 4.2.

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

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

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

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

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

The connection manager contains all GATT-related communication logic. Each device has exactly one GATT server and an arbitrary number of GATT client instances, one for each device it is connected to. The GATT server implements a single GATT service containing two characteristics which are shown in Figure 4.3. The **Public Key** characteristic has a *readable* permission and simply contains a public key of the peer. It is used to determine the identity of the device when initializing the connection and can be used for authentication and encryption in the further communication. The **Writer** characteristic with a *writable* permission is then used for sending data from the client to the server. As we want to support bidirectional communication, every two devices need to have a pair of client–server and server–client connections. This can be implemented in a way that every time a GATT server receives an incoming connection, the connection manager initiates an outgoing connection to the GATT server of the connecting device. Only after both links are established, the connection is considered ready.

4.3. Relay Protocol with Bandwidth Accounting

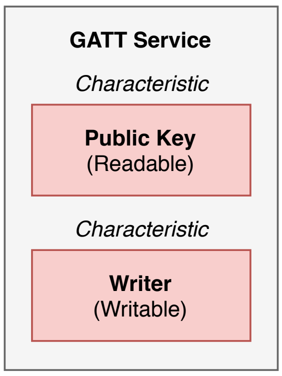


Figure 4.3: Our GATT Server Architecture

5 Implementation

6 Experiment

6.1. Analysis and Puncturing of Carrier Grade NAT

According to the report by Statista [3], there were three major mobile phone operators providing services in the Netherlands in Q4 2018. They are listed in Table 6.1. 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.

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

Table 6.1: 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. [3]

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 [2]. 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, as NAT behavior can change over time, we performed some more tests to verify that 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 Vodafone network changes the mapped port for new connections approximately every 60 seconds, even when connecting to the same IP address and a different port. This behavior can be described as *Address and Port-Dependent Mapping,* 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 [6][4].

6.2. Bootstrap Performance Evaluation

6.3. Stress Test

7 Conclusion

7.1. 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] 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`.

[3] 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/`.

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

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

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