HIGHER TECHNICAL FEDERAL TEACHING INSTITUTE Vienna 16
Department of Information Technology

DIPLOMA THESIS

LinkWave

Carried out in the school year 2023/24

Arshia Reisi (5CHITN) Supervisor: ***

Jan Schäfer (5AHITN) Supervisor: ***

Ruben Zukić (5AHITN) Supervisor: ***

Declaration of Independence

We declare that we have written the present diploma thesis independently and without external assistance, have not used any sources or aids other than those specified, and have identified passages taken literally or in substance from the sources used.

Vienna, on 04.06.2024

Authors:

Ruben Zukić

Jan Schäfer

Arshia Reisi

Documentation of the Diploma Thesis

Authors	Arshia Reisi Jan Schäfer Ruben Zukić
Class / School Year	*** / 2023-24
Topic	Development of a cross-platform application to optimize data and information transfer between devices with different operating systems

Submission Note	Date: 04.06.2024	Received by: ***
Approval	Date: 10.06.2024	Examiner: ***
	Date: 10.06.2024	Department Head: ***

Abstract

Have you heard about AirDrop? AirDrop is a feature of Apple operating systems that makes it possible to share data quickly and easily between Apple devices. Pictures and files can be sent to other devices at the touch of a button, and even the clipboard of your own devices is synchronized in the background.

LinkWave takes this functionality from Apple devices and extends it to an operating system-independent level. LinkWave is an innovative software solution that enables AirDrop-like connectivity across a wide range of platforms, including Windows, Android, and Apple devices themselves. The application enables the seamless sharing of files, images, and clipboard content across different operating systems without users having to worry about device compatibility.

Contents

Declaration of Independence ii

Documentation of the Diploma Thesis iii

Abstract v

1 Introduction 2

1.1 Problem Statement . . . 3

1.2 Solution . . . 5

2 Functionality 6

2.1 Functions . . . 7

2.2 Functionality on Different Operating Systems . . . 8

3 Applications 10

3.1 iOS . . . 12

3.2 macOS . . . 14

3.3 Android . . . 16

3.4 Windows . . . 19

3.5 Web - Dashboard . . . 22

4 Architecture 24

4.1 Selected Technologies . . . 25

4.1.1 WinUI . . . 25

4.1.2 Windows App SDK . . . 26

4.1.3 SwiftUI . . . 27

4.1.4 Jetpack Compose . . . 29

4.1.5 SvelteKit . . . 30

4.1.6 GraphQL . . . 31

4.1.7 Bonjour . . . 34

4.2 Network Architecture . . . 36

4.3 Software Architecture 37
4.3.1 iOS 37
4.3.2 macOS 39
4.3.3 Android 41
4.3.4 Windows 49
4.3.5 Server 55

5 Implementation of Features **60**
5.1 Device Discovery 61
5.1.1 Functionality 62
5.1.2 On iOS and macOS 64
5.1.3 On Android 68
5.1.4 On Windows 75
5.1.5 On the Server 77
5.2 File Transfer 81
5.2.1 Functionality 82
5.2.2 On iOS and macOS 84
5.2.3 On Android 87
5.2.4 On Windows 97
5.3 Clipboard 103
5.3.1 Functionality 104
5.3.2 On iOS and macOS 106
5.3.3 On Android 109
5.3.4 On Windows 114
5.4 User Account 118
5.4.1 Functionality 119
5.4.2 On iOS and macOS 129
5.4.3 On Android 131
5.4.4 On Windows 133
5.4.5 On the Web 136
5.4.6 On the Server 139

5.5 Background Execution 143
5.5.1 On iOS 144
5.5.2 On macOS 144
5.5.3 On Android 145
5.5.4 On Windows 147
5.6 Encryption 148
5.6.1 Functionality 149
5.6.2 On iOS and macOS 152
5.6.3 On Android 155
5.6.4 On Windows 158
5.6.5 On the Server 160

6 Conclusion **162**

List of Figures **164**

List of Tables **165**

Bibliography **165**

1 Introduction

The modern world's workday is becoming faster, more mobile, and more digital. Thanks to this development, it is now possible to work from anywhere in the world, at any time, as desired. To experience this flexibility, devices such as computers, smartphones, and tablets have become indispensable tools for everyday work. They allow important data and information to be accessed, edited, and shared anytime and anywhere.

1.1 Problem Statement

Many people now use multiple devices to complete their work—often utilizing devices from many different manufacturers. With the increasing use of various devices, it becomes more complicated to use them efficiently together.

Sharing data and information between devices, in particular, is often cumbersome and time-consuming. To illustrate the specific problems that frequently arise, some examples are given below:

- **Cumbersome File Transfer:** Transferring files between different devices, especially from different manufacturers, is cumbersome and time-consuming. For instance, transferring a file from an iPhone to a Windows computer requires relying on cloud services like Dropbox or Google Drive.

  This involves multiple steps, such as signing up for a cloud service, uploading the file via a web browser, signing in to the cloud service on the other device, and downloading the file. Additionally, programs unrelated to file transfer may need to be downloaded. As a result, file transfer is often so tedious that users avoid it altogether and restrict themselves to one device.

- **Transferring Large Files:** Especially in creative industries, large files like images, videos, and music—often several GB in size—need to be exchanged between different people and devices. Smaller teams and individuals often resort to cloud services like Dropbox or Google Drive to share these files.

  However, they are limited by internet speed and storage space. Additionally, files must first be uploaded and then downloaded, doubling the strain on already limited internet connections. Purchasing adequate storage from a cloud service becomes a significant cost factor, particularly for smaller teams and individuals.

- **Frequent File Transfers:** Even with smaller files, frequent uploading and downloading via cloud services quickly becomes tedious with repetition.

  This repetitive task can become so time-consuming that users limit their work to a single device, reducing flexibility and forcing them to find workarounds.

- **Transferring Text and Images:** When using multiple devices for work, transferring text and images between them is often necessary. This often involves cumbersome methods like emailing or sending text and images to oneself via messaging services like WhatsApp or Signal.

  For quick text transfers, such as weblinks, this is often too cumbersome. Even slight delays in transferring such frequently used data can quickly become extremely annoying over time.

Some manufacturers, like Apple, offer solutions within their ecosystem[1] such as “Airdrop” and “Universal Clipboard,” enabling seamless data sharing within their product range. However, a similarly integrated solution that works independently of manufacturer and operating system is missing. This is where LinkWave steps in, aiming to close this gap and provide a universal, user-friendly solution.

1.2 Solution

LinkWave aims to create an ecosystem experience across Mac, iPhone, Android, and Windows devices. LinkWave can be installed as an app on these devices. The app automatically detects other nearby devices with LinkWave installed. All LinkWave features can then be used between these devices.

[1] Ecosystem—in the context of technology, an ecosystem refers to the entirety of devices, software, and services provided by a single manufacturer.

2 Functionality

2.1 Functions

LinkWave offers a variety of features that enable the exchange of data and information between different devices. To provide a general overview of its functionality, the LinkWave features are briefly presented below:

- **Device Detection:** LinkWave automatically detects other nearby devices that also have LinkWave installed. For example, files can be transferred between a smartphone and a computer without manually establishing a connection.

- **File Transfer:** LinkWave allows files to be transferred between devices. It does not matter whether the devices are from different manufacturers or use different operating systems. The file transfer occurs via a direct connection between the devices without requiring an internet connection.

- **Clipboard Sharing:** LinkWave enables sharing the clipboard contents between devices. For instance, text or images copied on a smartphone can be pasted on a computer. This occurs automatically when the devices are nearby. However, for security reasons, it is only possible between devices logged in with the same user account.

- **User Account:** LinkWave allows users to log in with an account. This synchronizes the user's settings and data across all devices. For instance, the clipboard can be shared between devices logged in with the same user account.

- **Background Execution:** LinkWave can run in the background without the app being open. For example, files can be transferred between devices while the app runs in the background. Clipboard sharing is also possible in the background.

- **Encryption:** LinkWave encrypts all data transferred between communicating devices. This ensures that only the user can view the data. Encryption occurs automatically and is invisible to the user.

2.2 Functionality on Different Operating Systems

LinkWave supports a variety of operating systems. However, due to limitations of operating systems and hardware, not all features are fully functional on all operating systems. Table 1 lists the supported features on different operating systems and their limitations.

	Device Detection	File Transfer	Clipboard Sharing	User Account	Background Execution	Encryption
iOS	✓	✓	~	✓	✗	✓
macOS	✓	✓	✓	✓	✓	✓
Android	✓	✓	~	✓	✓	✓
Windows	✓	✓	✓	✓	✓	✓

✓ – Fully supported, ~ – Partially supported, ✗ – Not supported

Table 1: Supported Features on Different Operating Systems

The specific limitations and implementations on different operating systems are described in Chapter 5 “Implementation of Features” starting on page 61 and in the corresponding subchapters for the individual features.

3 Applications

The functions of LinkWave are provided in the form of apps on various operating systems. On each supported operating system, the app was developed in a native programming language to ensure the best possible integration. The apps are available for download on the LinkWave website at `linkwave.org`.

3.1 iOS

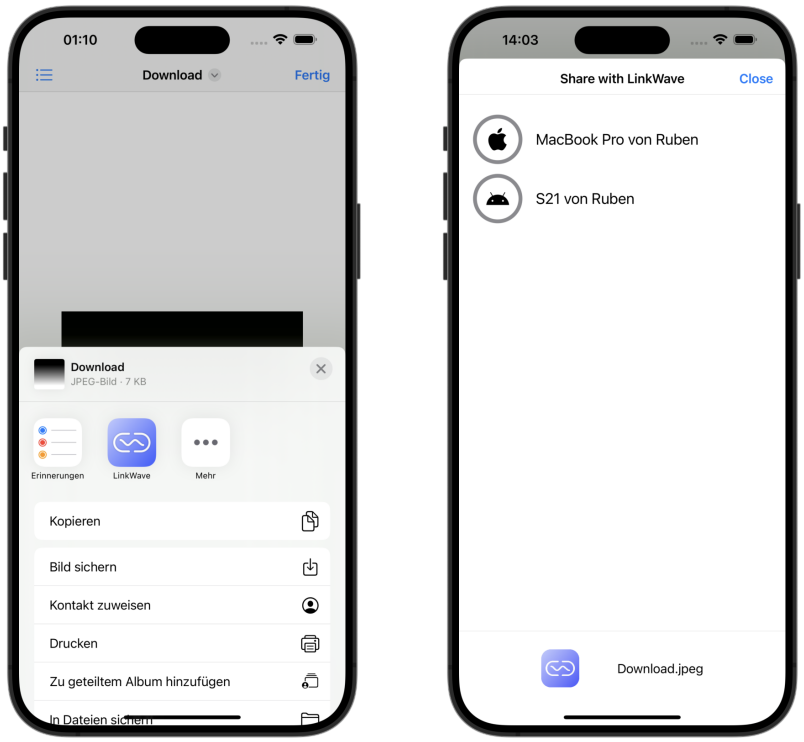
On iOS devices, the LinkWave app was developed in the Swift programming language. The UI was created using the SwiftUI framework. Swift and SwiftUI were developed by Apple and make it easy to create an interface that meets Apple's criteria. The app is available for iOS 17 and later.

To make it as easy as possible for users to use LinkWave, the iOS app offers various ways to use LinkWave. The implemented options are:

- **The LinkWave App:** In the app, users can use file transfer and adjust the app's settings. Additionally, users can log in to their LinkWave account in the app to unlock more features.

Figure 1: iOS LinkWave App

- **The "Share" Window:** iOS has a "Share" feature. This function allows users to share files, texts, and images from one app to another. Users only need to press the share icon in an app and choose from a selection of apps to share the content. LinkWave is one of these apps. This means users don't need to open the LinkWave app to share files. They simply press the share icon in another app and select LinkWave. From there, files can be sent directly to other devices, and texts can be copied directly to the clipboard of another device.



(a) iOS Share Selection

(b) iOS LinkWave Share Window

Figure 2: iOS Share Extension

3.2 macOS

On macOS devices, the LinkWave app was also developed in the Swift programming language. The UI was also created using the SwiftUI framework. The app is available for macOS 14 and later.

Like the iOS app, the macOS app offers various ways to use LinkWave. However, due to more development possibilities on macOS, these features are much more extensive. The implemented features are:

- **The LinkWave App:** In the app, users can use file transfer and adjust the app's settings, just like on iOS. Users can also log in to their LinkWave account to unlock additional features. Additionally, a list of all nearby devices using LinkWave is available.

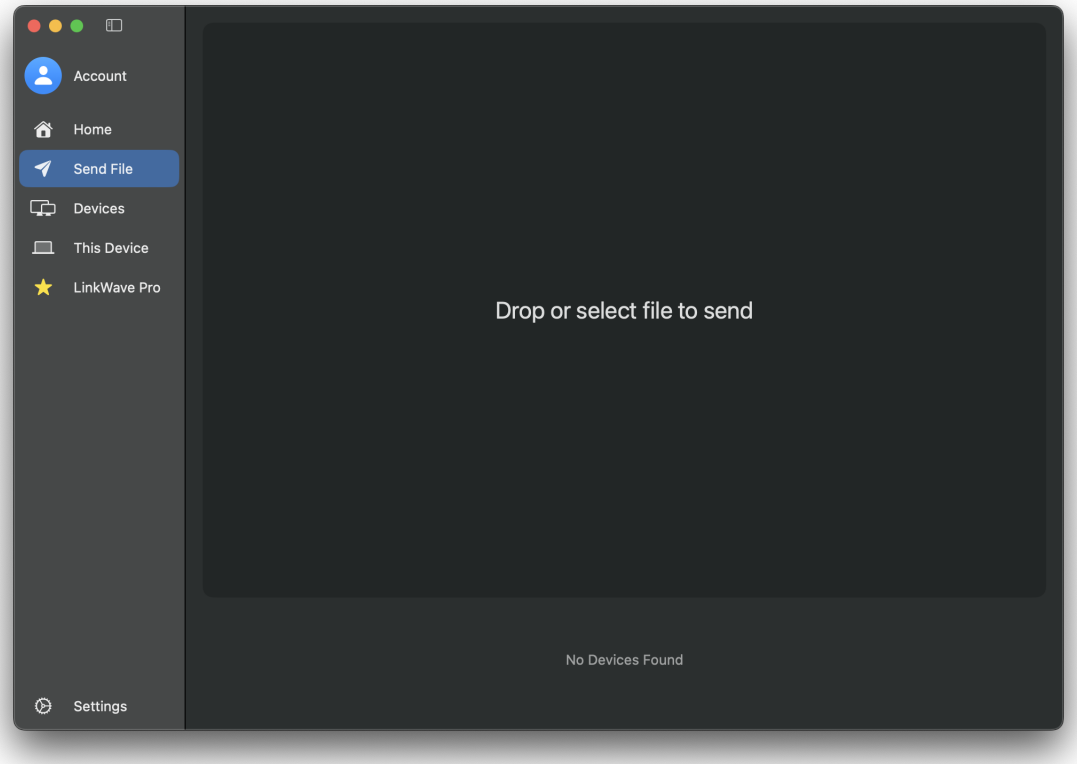
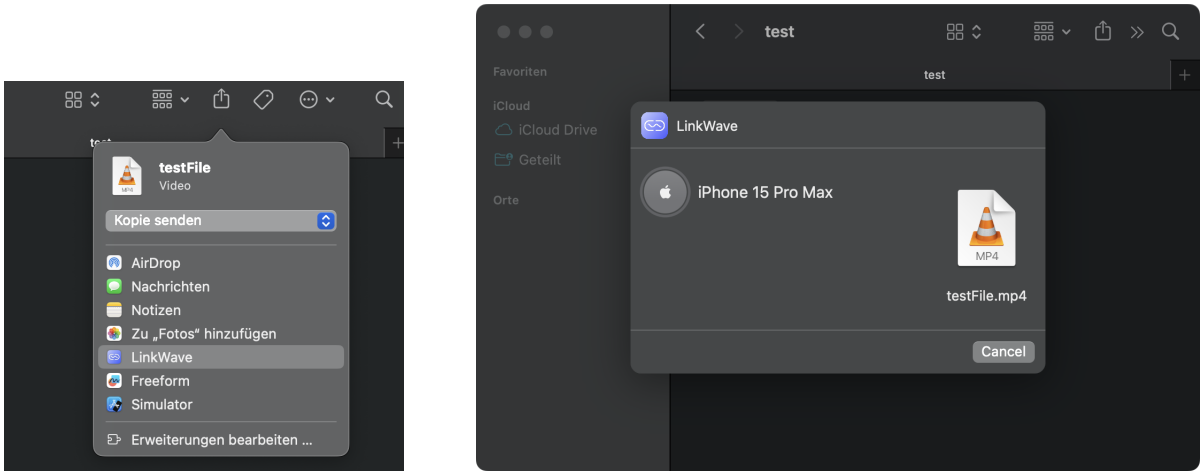


Figure 3: macOS LinkWave App

- **The "Share" Window:** macOS also has a "Share" feature. This works identically to the one on iOS. However, sharing text to the clipboard of another device is not possible on macOS. This is because the clipboard on macOS can be used directly and does not require a feature like on iOS.



(a) macOS Share Selection

(b) macOS LinkWave Share Window

Figure 4: macOS Share Extension

- **The Menu Bar Icon:** On macOS, there is a menu bar. In this menu bar, the icons of apps running in the background are displayed. LinkWave also has such an icon. Through this icon, users can open the LinkWave app and its settings. Users can view a list of devices connected to their account and access their settings directly. LinkWave can also be completely exited directly from here, so it no longer runs in the background.

Figure 5: macOS Menu Bar Icon

- **The Clipboard:** On macOS, as with any modern operating system, there is a clipboard. LinkWave automatically sends copied content to other devices connected to the same account.

3.3 Android

The LinkWave app for Android was developed in the Kotlin programming language. The UI was created using the Jetpack Compose framework. The app is available for Android 12 and later.

The Android app consists of three parts:

- **The LinkWave App:** The LinkWave app refers to the main application, the one that opens when you tap the app icon. In this app, users can adjust settings such as the visibility of their device and log in or out.

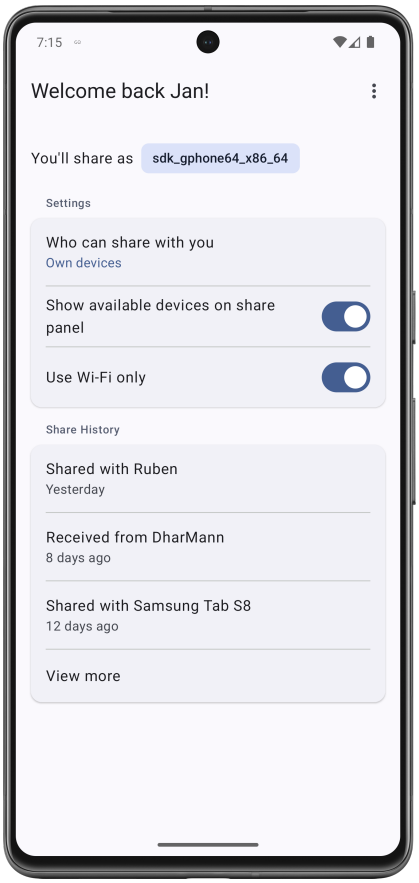
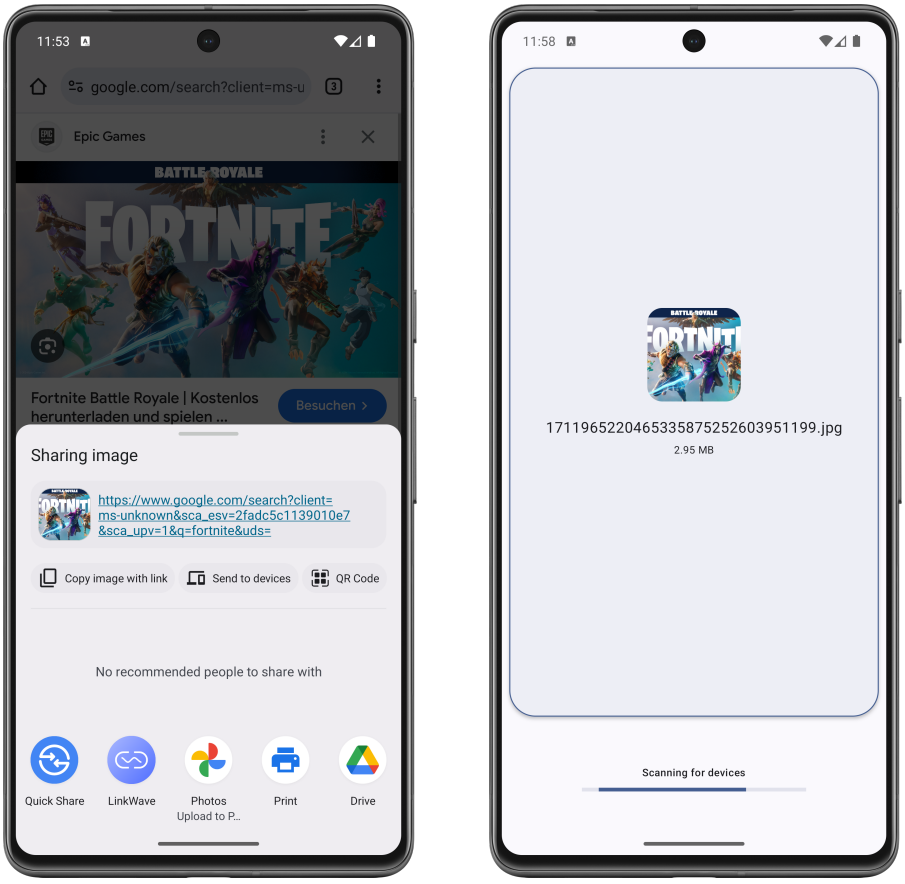


Figure 6: Android LinkWave App

- **The "Share" Window:** Android also has a "Share" feature. This feature works almost identically to the one on iOS. Images and files can be sent to the LinkWave app from other apps. To share an image or file, users simply need to tap the share icon in an app and then select the LinkWave app. This functionality is modeled after Android's native "Quick-Share" and Apple's "AirDrop."



(a) Android Share Selection

(b) Android LinkWave Share Window

Figure 7: Android Share Extension

- **The Clipboard:** Since Android 10, an app can only read from the clipboard when it is in the foreground. However, if text is selected, it can be sent to an app. To do this, users need to tap the three dots in the “Selection Menu” and then select the LinkWave app. The text is then sent to the LinkWave app, which forwards it to other devices. Figure 8 shows the Selection Menu.

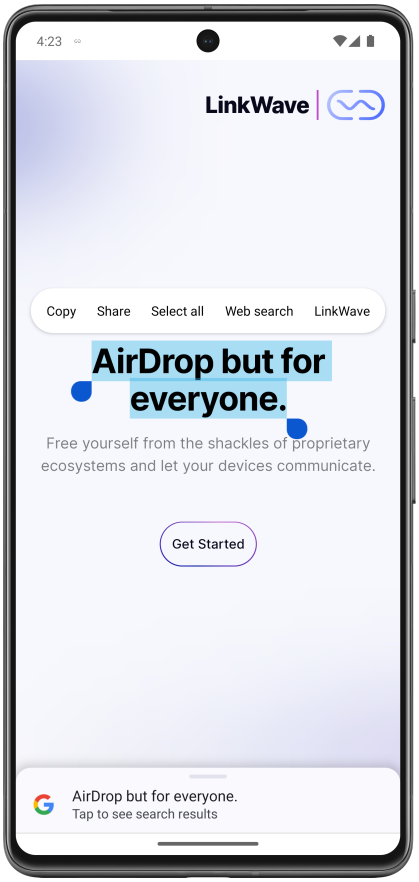


Figure 8: Selection Menu

3.4 Windows

The LinkWave app for Windows was developed in the C# programming language. The UI was created using the WinUI framework. The app is available for Windows 10 and later.

The Windows app offers similar usage options as the macOS app. The implemented features are:

- **The LinkWave App:** In the app, users can use file transfer and adjust the app's settings, just like on macOS and iOS. Users can also log in to their LinkWave account to unlock additional features. As on macOS, a device overview can also be accessed.

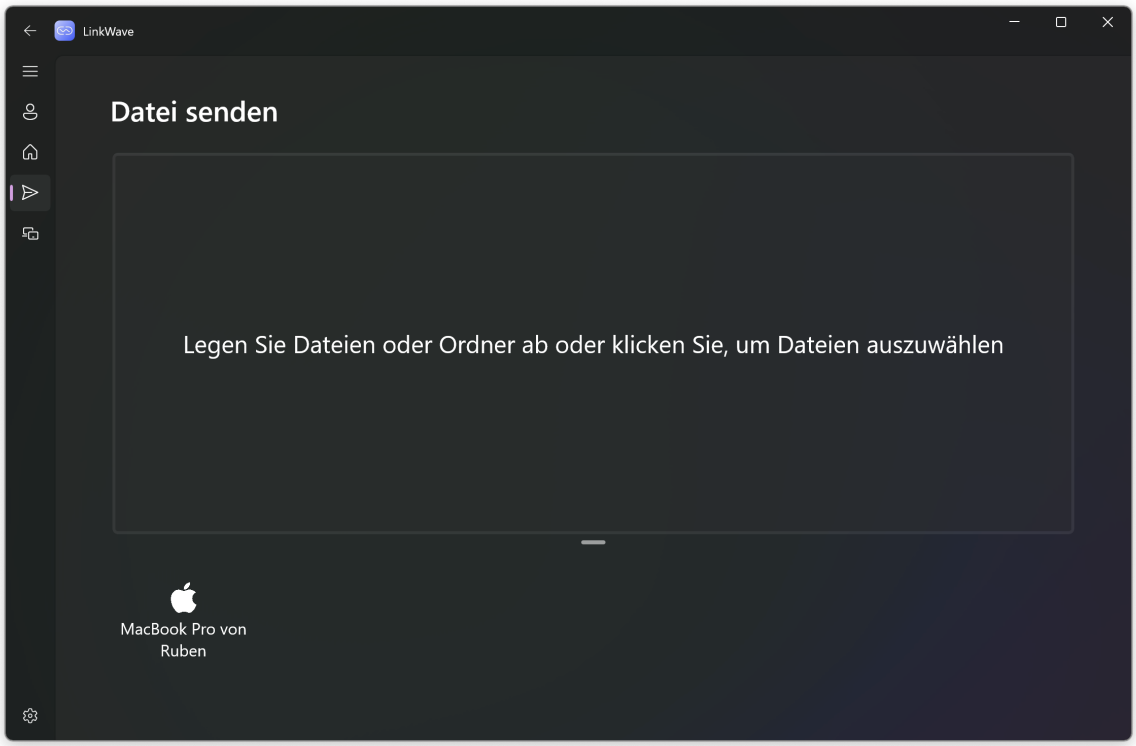


Figure 9: Windows LinkWave App

- **The “Share” Window:** Windows also has a “Share” feature. This feature works similarly to the one on macOS.

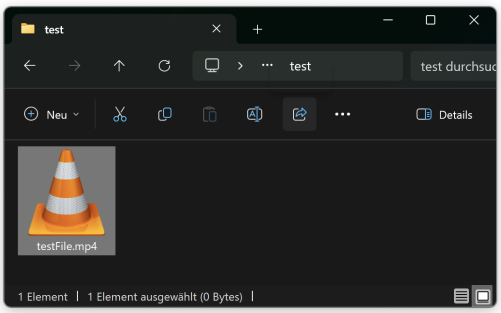
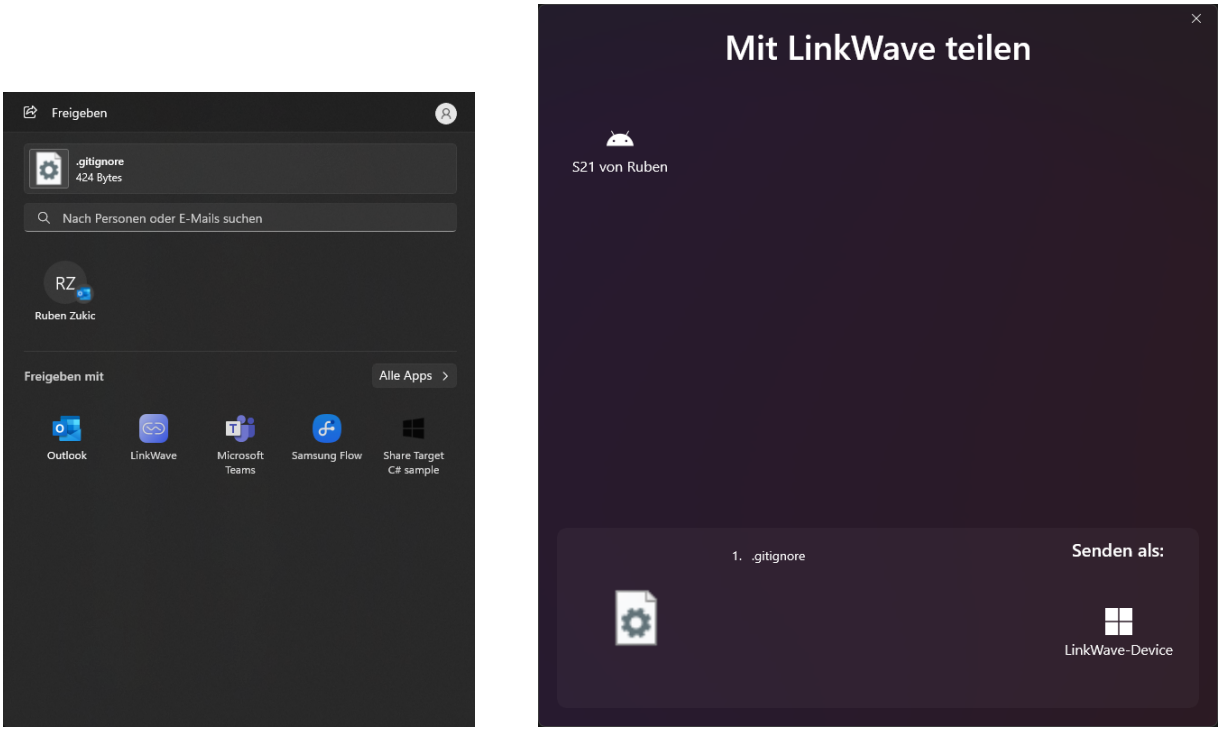


Figure 10: Windows Share in Explorer



(a) Windows Share Selection

(b) LinkWave Share Window

Figure 11: Windows Share Window

Users need to press the share icon in an app and will see a selection of apps to share the content with (see Figure 11 (a)). LinkWave is one of these apps. When selecting LinkWave, the LinkWave Share Window opens (see Figure 11 (b)). Here, users can choose which device to send the content to.

- **The System Tray[2] Icon:** Windows has a system tray. In this system tray, the

[2]The system tray is an area in the Windows taskbar where icons of apps running in the background

icons of apps running in the background are displayed. LinkWave also has such an icon (see Figure 12). Through this icon, users can open the LinkWave app or completely exit it so that it no longer runs in the background.

Figure 12: Windows System Tray Icon

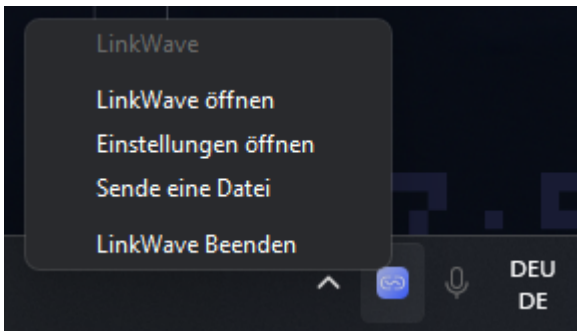


Figure 13: Windows System Tray Menu

- **The Clipboard:** As with any modern operating system, Windows also has a clipboard. LinkWave automatically sends copied content to other devices connected to the same account.

are displayed. It is located on the right side of the taskbar, identified by the ∧ symbol.

3.5 Web - Dashboard

The LinkWave Dashboard is a web application that allows users to manage their devices and general account-related settings. The dashboard was developed with SvelteKit and is available via browser on all devices. The web application is intended for managing one's account and devices. Users can view their devices and delete them if necessary. Account-related settings include options such as changing the password or deleting the account. The elements of the website were taken from the `shadcn/ui` UI library. The dashboard is accessible at `account.linkwave.org`.

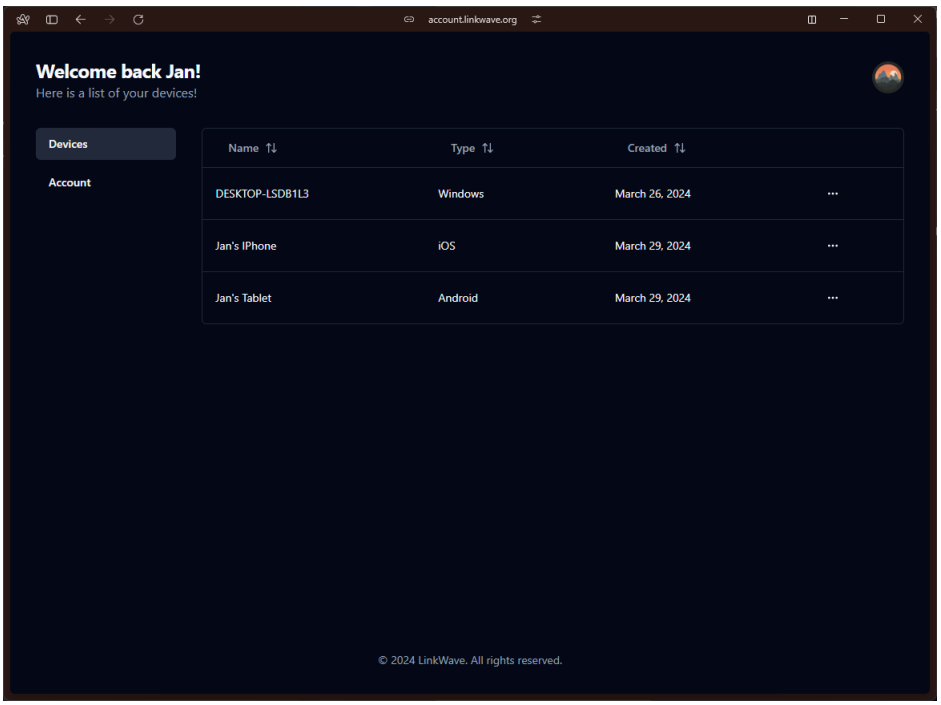


Figure 14: LinkWave Dashboard

4 Architecture

4.1 Selected Technologies

This section provides a detailed explanation of some key technologies and the reasons for their selection.

4.1.1 WinUI

Figure 15: WinUI Logo

What is WinUI?

WinUI is a UI library developed by Microsoft for creating modern user interfaces for Windows applications. These user interfaces use the same elements as Microsoft's system applications. [1] This design language is referred to by Microsoft as "Fluent Design." WinUI version 3 provides the elements from Windows 11 system applications, but they can also be used on older Windows versions (up to Windows 10). On these older versions, apps still appear as they do on Windows 11. The "Windows App SDK" allows developers to use these elements in their applications with C# and XAML. The "Windows App SDK" is described in more detail starting on page 26.

How is WinUI different from alternatives like MAUI?

.NET MAUI allows for creating similar user interfaces to WinUI. However, MAUI differs from WinUI in some respects. MAUI is a cross-platform UI library that enables the development of user interfaces for Windows, macOS, iOS, and Android. WinUI, on the other hand, is only available for Windows. Because MAUI must function across multiple operating systems, writing code that deeply integrates with the operating system is more difficult. WinUI, being built exclusively for Windows, does not have this limitation.

What was WinUI used for?

WinUI was used for developing the LinkWave app for Windows. The app was written in C# and XAML, which suited the needs of LinkWave.

4.1.2 Windows App SDK

What is the Windows App SDK?

“SDK” translates to software development kit. An SDK is a collection of tools that enables developers to create software for a specific platform. The “Windows App SDK” is an SDK from Microsoft that allows for the development of Windows applications. It helps create modern Windows apps using various UI libraries. For example, an app developed with the “Windows App SDK” could use either MAUI or WinUI.

The “Windows App SDK” also simplifies accessing Windows features. For instance, it allows easy access to the clipboard or displaying notifications.

How is the Windows App SDK different from other SDKs like UWP?

The “Windows App SDK” is an evolution of UWP, an older SDK from Microsoft serving a similar purpose. However, UWP is only available for Windows 10, whereas the “Windows App SDK” supports Windows 10 and later versions. It also enables the development of apps for Windows 11. Unfortunately, the “Windows App SDK” does not support some features that UWP does, such as easy integration of Google account sign-ins. Nevertheless, it makes sense to choose the “Windows App SDK” because UWP is already marked as “deprecated” by Microsoft and, unlike the “Windows App SDK,” no longer receives updates.

What was the Windows App SDK used for?

The “Windows App SDK” was used for developing the LinkWave app for Windows. The app was written in C# and XAML. With the “Windows App SDK,” it was possible to develop the app to function on Windows 10 and later while supporting the latest Windows 11 features. It also significantly simplified tasks like accessing the clipboard or sending notifications.

4.1.3 SwiftUI

Figure 16: SwiftUI Logo

What is SwiftUI?

SwiftUI is a framework developed by Apple for creating user interfaces for macOS (e.g., MacBooks) and iOS (e.g., iPhones). With SwiftUI, developers can use the same elements that Apple uses in its system applications.

How is SwiftUI different from alternatives like UIKit?

SwiftUI is a simpler and more modern version of UIKit. SwiftUI is declarative, meaning developers only need to describe how the user interface should look, not how it should be built. This makes creating user interfaces easier. SwiftUI is also easier for beginners to learn because it requires less code for the same functionality. An example of the difference between SwiftUI and UIKit can be seen in the code for a simple UI element:

SwiftUI enables faster development of user interfaces thanks to shorter and simpler code.

What was SwiftUI used for?

SwiftUI was used for developing the LinkWave app for macOS and iOS. The app utilizes special Apple elements, making it look and function as though it is an integral part of macOS and iOS themselves.

```
class ViewController:
  UIViewController {
  override func viewDidLoad() {
    super.viewDidLoad()
    let label = UILabel()
    label.text = "Hello, UIKit!"
    label.font = UIFont
      .systemFont(ofSize: 36)
    label.textColor = .blue
    view.addSubview(label)
  }
}
```

UIKit Example

```
struct ContentView: View {
  var body: some View {
    Text("Hello, SwiftUI!")
      .font(.largeTitle)
      .foregroundColor(.blue)
  }
}
```

SwiftUI Example

4.1.4 Jetpack Compose

Figure 17: Jetpack Compose Logo

Jetpack Compose is Google's recommended toolkit for building native user interfaces on Android, emphasizing a declarative approach to UI development. Unlike the traditional separation of XML layouts and Kotlin/Java code, Compose allows developers to programmatically create entire UIs using Kotlin. It simplifies and accelerates UI development on Android with less code, powerful tools, and intuitive Kotlin APIs. Kotlin, developed by JetBrains, is a programming language fully interoperable with Java. Many Android APIs, including the one used by LinkWave for the `Network Service Discovery Protocol` (see Chapter 5.1.3), are provided via Java code. Kotlin's interoperability with Java makes it possible to access these APIs from Jetpack Compose.

User Interface Unlike in WinUI or on the web, the user interface is not defined in a markup language such as `XAML` or `HTML`, but dynamically generated in code using so-called `@Composable` functions. Google provides a library of Composables corresponding to the "Material Design 3" design language used in Android system applications. The following example demonstrates how a simple component is constructed:

```
@Composable
fun Greeting(name: String) {
Text(text = "Hello␣$name!")
}
```

The `Greeting` function defines a `Text` element displaying the text "Hello" and the provided name.

Why was Jetpack Compose selected?

Jetpack Compose was used to develop the LinkWave app for Android. It utilizes Google's specific components, making it look and function as though it is a native part of Android itself. Unlike other technologies such as React Native or Flutter, which use an abstraction layer to communicate with Android, Jetpack Compose interacts directly with Android. This helps seamlessly integrate the app into the operating system.

4.1.5 SvelteKit

Figure 18: Svelte Logo

SvelteKit is a modern framework for developing web applications, built on the Svelte library. SvelteKit simplifies the development process by providing a structured environment for creating fast and reactive Single-Page Applications (SPAs) or Server-Side Rendered (SSR) applications. Unlike traditional frameworks, where most work is done in the user's browser, Svelte compiles code into highly optimized, imperative JavaScript during the build process, resulting in significant performance improvements. SvelteKit offers out-of-the-box support for routing, data prefetching, and seamless integration with various backend systems, enabling developers to focus on feature development rather than boilerplate code.

What was SvelteKit used for?

SvelteKit was used to develop the LinkWave Dashboard. The dashboard is a web application allowing users to manage their devices and change general account-related settings.

4.1.6 GraphQL

GraphQL is a query language for APIs, where an API (Application Programming Interface) is an interface allowing different software applications to interact and exchange functions or data. With GraphQL, developers define a schema that describes all available data and how to access it. Clients can specify in a single request exactly which data they need. The server interprets this request, retrieves the required data, and returns it in the format defined by the query. GraphQL views the information being queried as a connected system of nodes and edges, enabling the construction of deeply nested and complex queries. This approach allows targeted and efficient data retrieval tailored to the clients' needs.

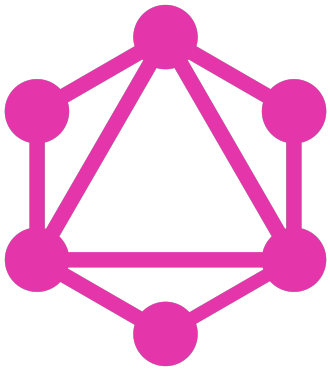


Figure 19: GraphQL Logo

Advantages of GraphQL:

- **Precise Data Requirements:** Allows users to request exactly the data they need through a single, unified interface.
- **Strongly Typed Schema:** Ensures precise validation and supports clear documentation through a strongly typed schema.
- **Improved Development:** Provides introspection capabilities that simplify frontend application development.
- **Minimized Data Transfer:** Reduces the amount of data transferred, which improves performance, particularly on mobile platforms.
- **Reduced Network Requests:** Decreases the need for multiple network requests, thereby improving the overall performance of the application.

Disadvantages of GraphQL:

- **Higher Query Complexity:** For new developers, learning GraphQL can be more challenging than REST due to its complexity and flexibility.
- **Caching Challenges:** Implementing client-side caching is more complicated because GraphQL queries are often specific and less predictable than REST queries.
- **Rate Limiting and Monitoring:** Monitoring and limiting query rates can be more difficult due to the variable nature of GraphQL queries.
- **Performance for Large Queries:** Large and complex queries can impact performance by consuming significant resources at once.

Advantages of REST APIs:

- **Simplicity and Understandability:** REST APIs follow a standardized approach with clear conventions, making it easier for developers to work with REST.
- **Wide Support:** REST is natively supported on many platforms and languages, simplifying integration into existing systems.
- **Efficient Caching:** REST enables effective caching of requests, reducing network load and server requests.
- **Statelessness:** Each client-to-server request is independent and contains all the information the server needs to respond.

Disadvantages of REST APIs:

- **Overfetching and Underfetching:** REST APIs can result in delivering too much or too little data, leading to inefficient requests.
- **Multiple Endpoints:** The need to define separate endpoints for various data resources can increase complexity.
- **Rigid Structures:** The rigid structures of REST can complicate the development of dynamic applications with flexible data requirements.
- **Challenges with Data Aggregation:** Merging data from different sources or endpoints can require additional requests and complexity.

4.1.7 Bonjour

The integration of Bonjour in LinkWave facilitates networking in an efficient and user-oriented manner. Thanks to Bonjour, LinkWave can quickly identify other devices equipped with LinkWave on the network.

Figure 20: Bonjour Logo

What is Bonjour?

The Bonjour protocol, also known as Zero Configuration Networking (Zeroconf), is a technology developed by Apple. Bonjour allows devices to recognize each other within a local network without requiring manual network configuration. The protocol is not only used by Apple operating systems but also by many other operating systems. [17]

How does Bonjour work?

Using Multicast DNS (mDNS), Bonjour provides a platform for devices to independently publish and discover their services within a local network. This technology is highly compatible across various operating systems, which is one of the main reasons why LinkWave uses Bonjour to locate other devices in a local network.

Service Discovery The Bonjour protocol utilizes **Multicast DNS (mDNS)**, a method for discovering network services that fundamentally differs from traditional DNS. Instead of relying on centralized servers, mDNS allows devices and services to independently publish and discover themselves within the local network. This is achieved by sending requests to a specific multicast IP address, which is shared and monitored by all network devices.

Name Resolution The Bonjour protocol simplifies name resolution within the network. Remembering IP addresses is no longer necessary. This is achieved through the use of Multicast DNS (mDNS), which automates the assignment of user-friendly names to IP addresses within the local network. Devices send multicast requests to the address 224.0.0.251 for IPv4 or ff02::fb for IPv6 to resolve a device name or discover services. Furthermore, this not only enables the identification of services on the network but also provides additional metadata to support precise service identification.

Automatic Configuration By automating network configuration, Bonjour significantly simplifies the setup of devices within the network. It manages IP address assignments via DHCP as well as in the APIPA range (169.254.x.x) if no DHCP server is available.

Broad Compatibility Although developed by Apple, Bonjour supports a variety of operating systems beyond macOS and iOS, including Windows (with Bonjour software installed) and other systems that implement mDNS and DNS-SD. This underscores its cross-platform applicability.

Zero-Configuration Networking (Zeroconf) The concept of “zero-configuration” implies that devices function with minimal to no user intervention. This is particularly advantageous for users without technical expertise or those who prefer not to configure complicated settings.

4.2 Network Architecture

LinkWave uses a client-server model to receive and process requests. Communication occurs via the TCP/IP protocol, which ensures secure and reliable data transmission. The server acts as the receiver, processing the data requests sent by clients.

The security of the data is ensured by the SSL/TLS protocol, which enables encryption and authentication of the communicating parties. At the beginning of each connection, a handshake occurs, during which certificates and cryptographic keys are exchanged to establish a secure connection. Incoming client requests are received by the server via TCP listeners and internally forwarded to the corresponding services responsible for tasks such as file or clipboard sharing. This process is explained in the following sections.

4.3 Software Architecture

4.3.1 iOS

The following section describes the software architecture of the iOS app for LinkWave. The iOS app was written in Swift using SwiftUI and, like the Windows app (see Chapter 4.3.4), follows the MVVM (Model-View-ViewModel) architectural pattern.

iOS Core Concepts

Every iOS app consists of multiple extensions that work together to provide the app's functionality. Each extension has a specific purpose and is independent of the others. Every iOS app has at least one main application that controls the app's user interface (see Chapter 3.1). Other extensions can either logically group code or provide functions that work without the main application. The following extensions were used in the iOS app for LinkWave:

- **App (LinkWaveApp):** The main application that controls the app's user interface. The exact implementation of the app is described in Chapter 3.1.

- **Framework (LinkWaveFramework):** A collection of classes and functions logically separated from the main application. Frameworks serve the same purpose as class libraries in other programming languages like C#. They allow other extensions that cannot access the main application's code to access functions and classes used by both the main application and other extensions. This framework is shared with the macOS app.

- **Share Extension (LinkWaveShare):** An extension that allows content from other apps to be shared with the LinkWave app. For example, users can share a file from the Files app to the LinkWave app to send the file to another device. This extension is independent of the main application and can function without it. However, it is installed alongside the main application and can access functions and classes from the framework.

LinkWaveApp in Detail The `LinkWaveApp` can be compiled into an app that can be published on the App Store. The main application of the iOS app for LinkWave is the LinkWaveApp. It consists of multiple views written in SwiftUI. Each view represents a screen within the app that users can see and interact with. This means that the entire user interface (UI) of the app is defined within the LinkWaveApp.

LinkWaveFramework in Detail The `LinkWaveFramework` contains classes and functions used by the LinkWaveApp and the LinkWaveShare extension. It includes the logic required for communication with other devices on the network and the management of files and the clipboard. The framework is written in Swift.

LinkWaveShare in Detail The `LinkWaveShare` extension allows content from other apps to be shared with the LinkWave app. It consists of a view that enables users to share content and a class that contains the logic for sharing the content. The extension is written in Swift and uses the LinkWaveFramework to enable communication with other devices. The extension is described in more detail in Chapter 3.1.

4.3.2 macOS

The following section describes the software architecture of the macOS app for LinkWave. The macOS app was written in Swift using SwiftUI and, like the Windows app (see Chapter 4.3.4), follows the MVVM (Model-View-ViewModel) architectural pattern.

macOS Core Concepts

SwiftUI apps on macOS function exactly like SwiftUI apps on iOS. The main difference is that macOS apps run on a Mac, whereas iOS apps run on an iPhone or iPad. The architecture of the macOS app for LinkWave is identical to the iOS app, except that the user interface is tailored to macOS.

Additionally, macOS apps are much easier to publish than iOS apps. While iOS apps can only be published on the Apple App Store, macOS apps can also be distributed via a download link as a .dmg file. This allows macOS apps to be distributed without the App Store.

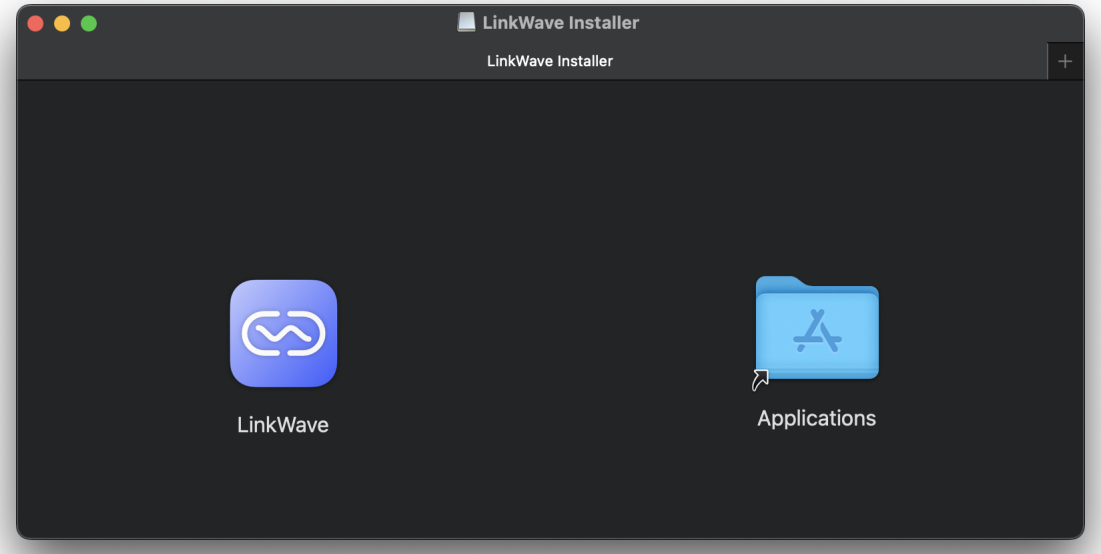


Figure 21: macOS .dmg Installer

The macOS app for LinkWave was written in Swift using SwiftUI and is structured in the same way as the iOS app (see Chapter 4.3.1). It consists of the following three components:

- `LinkWaveApp`: Responsible for the app's user interface.
- `LinkWaveFramework`: Contains logic shared between the app and the share extension.
- `LinkWaveShare`: Enables sharing content from other apps to the LinkWave app.

These components have essentially the same functionality as in the iOS app. However, under macOS, the `LinkWaveApp` has been adjusted.

LinkWaveApp in Detail The `LinkWaveApp` is responsible for the app's user interface. What differentiates the `LinkWaveApp` on macOS from the one on iOS is that macOS supports more than one window and additional features like an icon in the menu bar. These features are located in the `LinkWaveApp`, as they should only be available when the app is open.

This means that, in addition to the view for the app's main window, views for the menu bar icon and the settings window are also defined in the `LinkWaveApp`. The following views are present in the `LinkWaveApp`:

- **Window(id: "main")**: The main window of the app (see Chapter 3.2). This window includes a `NavigationSplitView` with a variable `DetailView`, which changes based on the user's page selection. This view allows the app to have a sidebar with different pages.
- **Window(id: "permissions")**: A window that shows the user what permissions the app requires.
- **MenubarExtra**: The menu bar icon that allows the user to open and close the app.
- **Settings**: A window where the user can change the app's settings.

4.3.3 Android

The following section describes the software architecture of the Android app for LinkWave. The Android app was written in Kotlin using Jetpack Compose and, like the Windows app (see Chapter 4.3.4), follows the MVVM (Model-View-ViewModel) architectural pattern.

Android Core Concepts

Every Android app consists of multiple components that work together to provide the app's functionality. The key components are briefly introduced below.

Activities An Activity can be understood as a screen within an app. Each screen with which a user interacts (e.g., login screen, home screen, settings screen) is typically managed by its own Activity. An Activity represents the window in which the app draws the UI (user interface). Each Activity is independent of the others and can launch other Activities using so-called Intents. The following example illustrates an Activity in Kotlin:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Greeting("Android")
                }
            }
        }
    }
```

Intents Intents are messaging objects used to request an action from another app component. This could involve starting another Activity or Service or using Android features like opening a webpage. Intents can also be used to transfer data between components. The following example shows an Intent that opens the YouTube app:

```
val intent = Intent(Intent.ACTION_MAIN).apply {
    `package` = "com.google.android.youtube"
}
startActivity(intent)
```

Here, a `MAIN` Intent is created and sent to the package `com.google.android.youtube`. Android provides a variety of Intents to perform different actions. For example, a file can be sent to another app using a `SEND` Intent. The `MAIN` Intent starts the `MainActivity` of an app. The Intents an app supports are defined in the `AndroidManifest.xml` file.

Services A Service is a component that runs in the background to perform long-running operations. For example, a Service can play music in the background without the user having the app in the foreground. LinkWave uses a Service to announce the device on the network in the background. The following example shows a Service:

```
class LinkWaveService : Service() {
    override fun onStartCommand(
        intent: Intent,
        flags: Int, startId: Int
    ): Int {
        startForeground(1, notification)
        startDiscovery()
        return super.onStartCommand(intent, flags, startId)
    }
}
```

Since Android API Level 26 (Android 8.0), Services must run in the foreground, meaning they must display a notification to inform the user that the Service is running in the background and consuming resources. Such Services are referred to as `Foreground Services`.

Broadcast Receiver Broadcast Receivers allow the app to receive broadcasts from other apps or the system. An example would be listening for a broadcast indicating that the device's battery is low or that the device has been restarted.

```
class BootCompletedReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        if (Intent.ACTION_BOOT_COMPLETED == intent.action) {
            // Code to execute when the boot is completed
        }
    }
}
```

Content Provider Content Providers enable data sharing between apps. A Content Provider provides a standardized interface for accessing data. An example would be sharing contacts between different apps or listing all media files (e.g., images and videos) on the device.

AndroidManifest.xml The `AndroidManifest.xml` is the configuration file where app components are configured. It defines which components exist, which permissions the app requires, and which Intents the app supports. It also specifies which Activity is opened when the app starts.

Trailing Lambdas in Kotlin Kotlin allows passing functions as the last argument of another function, referred to as trailing lambdas. If a lambda expression is the last parameter of a function, it can be placed outside the parentheses of the function call. This syntax is frequently used in Jetpack Compose to make UI code more structured and declarative.

```
@Composable
fun FancyBox(content: @Composable () -> Unit) {
    Box {
        content()
    }
}
```

The example shows a `FancyBox` function that creates a `Box` element with any given content. The content is passed as a lambda expression. This component can then be used as follows:

```
FancyBox {
    Button(onClick = { /*TODO*/ }) {
        Text(text = "Click me!")
    }
}
```

`Button` is another composable function that expects two parameters: `onClick` and `content`. `onClick` is a function executed when the button is clicked. `content` is the button's content, which in this case is a text element.

Dependency Injection Dependency Injection (DI) is a design pattern that allows a program to be independent of how its dependencies are created, assembled, and represented. The goal is to remove hard-coded dependencies and allow them to be changed either at runtime or compile-time. This makes it possible to use mock objects in tests to replace dependencies. The DI framework Koin was used in the development of the LinkWave app.

```
class DevicesViewModel(
    private val bonjourController: BonjourController
) : ViewModel() { ... }
```

This ViewModel depends on a `BonjourController`, which is passed via the constructor. Koin ensures that the `BonjourController` is created at runtime and passed to the ViewModel.

```
val appModule = module {
    single { BonjourController() }
    viewModel { DevicesViewModel(get()) }
}
```

`get()` returns the `BonjourController` created by Koin at runtime. `single` means that the `BonjourController` is created only once, and each subsequent call to `get()` returns the same `BonjourController`.

Android Program Structure

As mentioned above, the Android app follows the MVVM architectural pattern. How the different layers of the MVVM pattern interact is shown in Figure 22. The OS layer (Operating System Layer) launches a specific Activity via an Intent. The Activity accesses data from the Data Layer through ViewModels, which form the Presentation Layer.

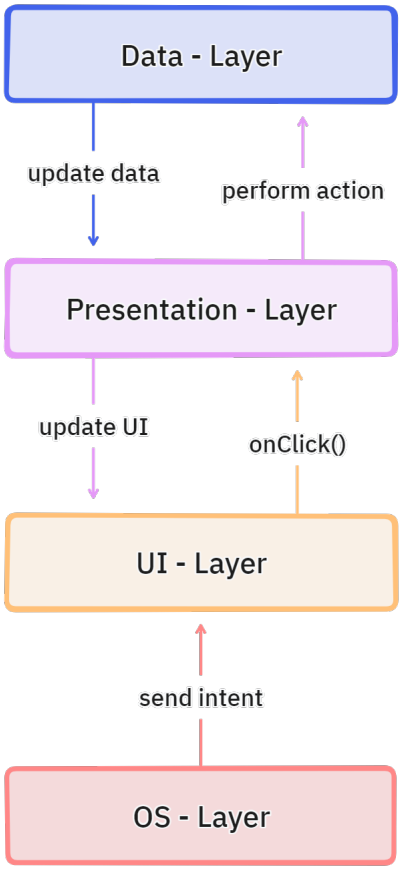


Figure 22: OS - Features

Activities Activities are part of the UI Layer. The app consists of three Activities:

- **MainActivity:** The `MainActivity` is the window that opens when the app's icon on the Android home screen, known as the launcher, is pressed. The launcher sends a `MAIN` Intent to the app, which then opens the `MainActivity`. In the `MainActivity`, users can adjust settings such as network visibility and log in or out.

- **ShareActivity:** The `ShareActivity` opens when a `SEND` Intent is sent to the app. This happens when a user uses Android's "Share" feature to share a file. In the `ShareActivity`, the user can select a device to which the file will be sent. The selected device is then sent to the `FileSendManager`, which handles the file transfer. The `FileSendManager` provides information about the transfer status, which is passed to the UI Layer via a ViewModel.
- **ClipboardActivity:** The `ClipboardActivity` opens when a `PROCESS_TEXT` Intent is sent to the app. Unlike the `ShareActivity` and `MainActivity`, the `ClipboardActivity` does not display a UI. It simply passes the selected text to the `ClipboardShareManager` via a ViewModel. The `ClipboardShareManager` then sends the text to all devices associated with the same account.

Data Data required for each Activity can be accessed via ViewModels. ViewModels are part of the Presentation Layer of the app. They contain the logic needed to process the data from the Data Layer and pass it to the UI components. For example, to display the list of discovered devices in the `MainActivity`, the `DevicesViewModel` is used. The `DevicesViewModel` contains a list of devices discovered via the `BonjourController`. This list is passed to the `MainActivity`, which then displays it. This data flow is implemented using Kotlin Flow [6].

LinkWaveService Since some functions need to be available in the background, a foreground service is required. Even if the app is running in the background or closed, the device should remain visible on the network and accept requests, such as sharing a file. The `LinkWaveService` is a foreground service that automatically starts when the device is rebooted or the app is opened. The service instantiates and starts the `BonjourAdvertiser`, which ensures the device is discoverable on the network, and a `RequestListener`, which handles TCP requests such as file transfers or clipboard sharing. This allows the device to handle requests even when the app is closed. Figure 23 shows the UML diagram of the LinkWaveService.

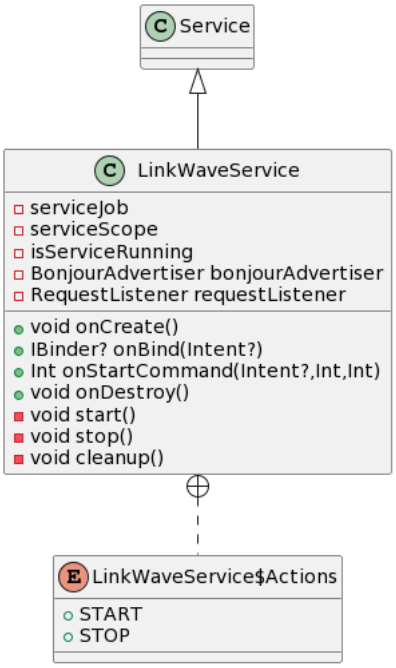


Figure 23: UML of LinkWaveService

4.3.4 Windows

MVVM

This section describes the software architecture of the Windows app for LinkWave. The Windows app was written in C# and XAML. The app follows the MVVM (Model-View-ViewModel) architectural pattern. MVVM is a design pattern that separates the user interface from the logic. This makes the logic easier to test and maintain. The MVVM architecture consists of three parts:

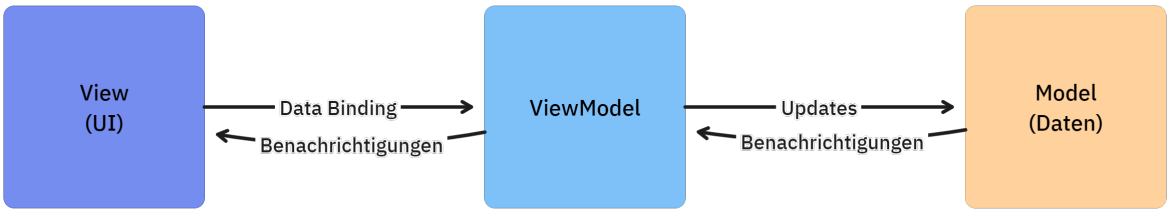


Figure 24: MVVM Architecture

- **Model:** The Model is the data structure that stores the app's data. The Model contains no logic, only the data. In the Windows app for LinkWave, the Model includes, for example, user data and device data.
- **View:** The View is the user interface of the app. The View displays the Model's data and forwards user input to the ViewModel. In the Windows app for LinkWave, the View includes, for example, the window where users can initiate file transfers.
- **ViewModel:** The ViewModel contains the app's logic, which processes the Model's data and passes it to the View. The ViewModel also contains the logic that processes user input and passes it to the Model. In the Windows app for LinkWave, the ViewModel includes, for example, the logic that starts file transfers.

Windows C# Project Structure

Microsoft recommends creating WinUI 3 projects using the Visual Studio extension "Template Studio for WinUI (C#)." This extension generates a project with a precon-

figured structure consisting of the following components:

- **The UI Project:** The UI project is a WinUI application that contains the app's user interface. The app is built from this project. All other projects only include code used by this project.
- **The Core Project:** The Core project is a class library that contains the core logic of the app. It includes utility functions used by the user interface, simplifying the use of functions from the backend project. The Core project itself does not include any functions that directly access the network or file system.
- **The Backend Project:** The Backend project is a class library that contains the logic for directly accessing the network or file system. It includes functions for tasks such as file transfers or device discovery. The Backend project is used by the Core project to implement the app's core logic.

UI Project in Detail

The UI project can generate a “Packaged App” in the form of an `.msix` file. This app can be distributed and installed on any Windows system from Windows 10 onward. Installation is done by double-clicking the file, and it looks as follows:

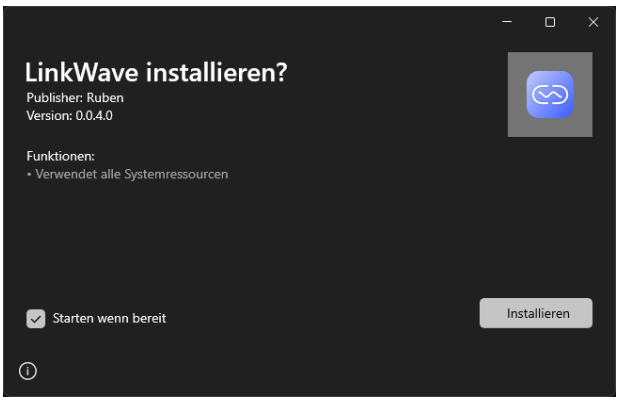


Figure 25: Packaged App Installation

This installs the app along with all its components on the device. This means that both the app itself and the “Share Extension” are installed.

The app consists of a single window (`MainWindow.xaml`). Within this window is a page containing the menu and a frame for the app’s various pages (`ShellPage.xaml`). The different pages are implemented as WinUI Pages (`<name>.xaml`) and displayed within the ShellPage. These pages include:

- **MainPage.xaml:** The main page of the app. This is displayed when the app is opened. It provides an overview of the app’s various functions and the user’s account.
- **AccountPage.xaml:** The account page of the app. Here, users can log in to their account and modify account settings.
- **DevicesPage.xaml:** The devices page of the app. This displays an overview of nearby devices using LinkWave.
- **SendFilePage.xaml:** The file transfer page of the app. Here, users can send files to other devices.

- **SettingsPage.xaml:** The settings page of the app. Here, users can modify the app's general settings.

The "Share Extension" is implemented as a sharing target and has its own window (`ShareWindow.xaml`). This window displays a single page (`SharePage.xaml`).

Core Project in Detail

The Core project contains the app's core logic. It includes utility functions used by the user interface, simplifying the use of functions from the Backend project. These functions are organized into utility classes (Services). The main classes include:

- **LinkWaveService:** The LinkWaveService manages all LinkWave functions, such as discovering nearby devices. The LinkWaveService handles everything the main app does with the Backend project.
- **LinkWaveShareService:** The LinkWaveShareService manages all LinkWave functions related to the Share Extension. It performs similar tasks to the LinkWaveService but only for functions used by the Share Extension. For instance, the LinkWaveShareService cannot receive files from other devices.

Backend Project in Detail

The Backend project contains the logic for directly accessing the network or file system. It includes functions for tasks such as transferring files or discovering devices. The Backend project is used by the Core project to implement the app's core logic.

The LinkWave.Backend DLL plays an essential role in the overall system, enabling the implementation of key services such as FileSharing, ClipboardSharing, secure encryption, and efficient caching of data.

Core Components of the LinkWave.Backend DLL

The LinkWave Receiver In the LinkWave.Backend architecture, the `LinkWaveReceiver.cs` component serves as the primary receiver for all incoming requests. This central interface receives requests and forwards them to `RequestSwitcher.cs` to invoke the appropriate functions.

The RequestSwitcher Another critical component is `RequestSwitcher.cs`, which plays a key role in routing requests to specific functions. It acts as a dispatcher, analyzing each request and forwarding it to the relevant function based on its content and purpose.

Using the RequestSwitcher, LinkWave.Backend can efficiently respond to various demands by directing requests to the designated processing units.

SSL Handshake The `SSL.cs` class is central to implementing the SSL handshake, establishing a secure and encrypted connection between client and server. This class ensures that all data transmissions are protected against unauthorized access.

Caching `SqliteHandler.cs` serves as the interface to the internal SQLite database, providing wrapper functions for data queries as well as adding and updating data. This class plays a crucial role in managing cache information and ensuring efficient data processing within the application.

File and Clipboard Sender/Receiver The classes `FileSender.cs`, `FileReceiver.cs`, `ClipboardSender.cs`, and `ClipboardReceiver.cs` are specifically designed for sending and receiving file contents or clipboard data. These classes enable seamless data exchange between users.

4.3.5 Server

Overview

The LinkWave Backend Service is designed as a central interface and plays a critical role in managing and securing the system. Its primary function is to authenticate users and devices to ensure that only authorized entities gain access. Additionally, it is responsible for data storage while maintaining high standards and best practices for data integrity and privacy.
LinkWave uses GraphQL as a query language for communication with its API. Special security measures in GraphQL ensure that sensitive data remains protected and is exclusively managed by the server. For password protection, LinkWave uses a robust algorithm called “Bcrypt.” Compared to other commonly used hash functions like SHA-256, Bcrypt offers greater protection against brute-force attacks by significantly slowing them down through its computationally intensive algorithm. This makes a significant contribution to user data security. LinkWave uses the PostgreSQL database system, known for its performance, to store data. The database design follows the code-first principle using the EF Core .NET framework, allowing flexible data structure development and simplifying database management and expansion.

Concept

In the backend architecture of LinkWave, we provide an efficient communication channel that remains consistent across multiple platforms. Interaction begins when the website, desktop app, or mobile app of the client sends a request. This request is then forwarded to the server, where it is processed via the API.

A key component of our server is the GraphQL API, designed for processing specific queries. The implementation of GraphQL uses Hot Chocolate, a library that facilitates the integration of GraphQL into .NET applications [9]. Hot Chocolate offers a variety of features that accelerate the development of GraphQL APIs and simplify their integration into existing .NET applications. This allows app users to query exactly the information they need, reducing the transmission of unnecessary data and optimizing system performance.

Once a request is processed by the GraphQL API, interactions with the PostgreSQL database occur. Here, CRUD operations—Create, Read, Update, and Delete—are executed. PostgreSQL was chosen for its reliability in storing and querying data.

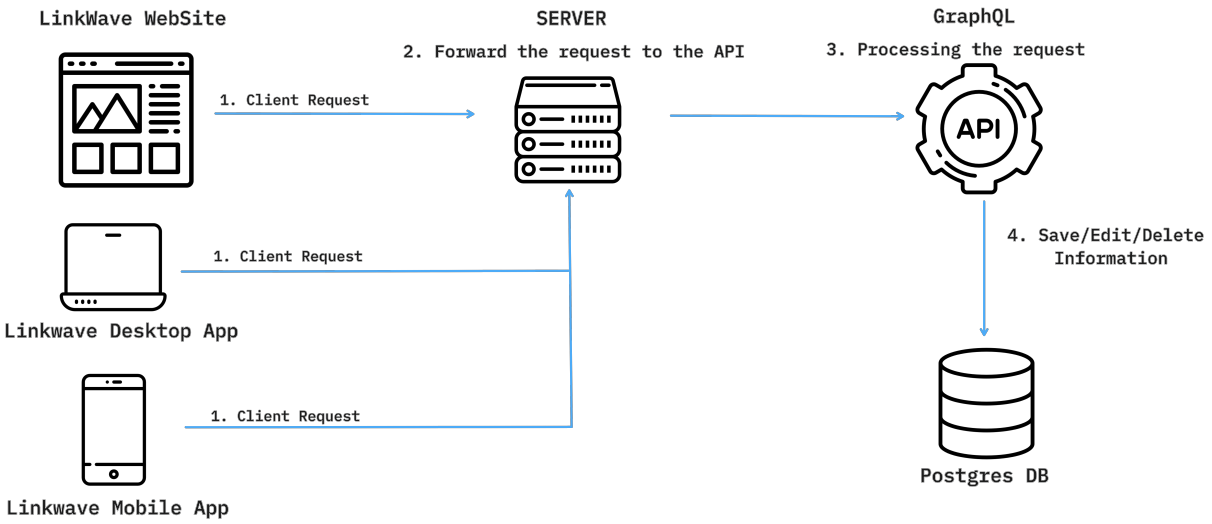


Figure 26: Backend Architecture

Server and Hosting

Microsoft Azure is used as the cloud platform for hosting. Docker Compose manages the Docker containers on this server, enabling efficient orchestration and scaling of our applications.

Nginx as Reverse Proxy

An essential part of the infrastructure is the use of Nginx as a reverse proxy. This acts as a proxy between user requests (e.g., via the web) and our server, receiving the requests and forwarding them to the appropriate running service in Docker containers. This provides several advantages:

- **Security:** Nginx hides the identity of the backend server, enhancing security by preventing attackers from obtaining direct information about the internal server.
- **SSL Encryption:** Nginx enables simple configuration of SSL/TLS to ensure secure communication between client and server.

The Nginx server for LinkWave is configured to redirect all HTTP requests to the secure HTTPS channel, as shown in the following snippet:

```
server {
    listen 80;
    server_name api.linkwave.org;
    return 301 https://$host$request_uri;
}
```

SSL Encryption

Only modern and secure SSL protocols and encryptions are used to ensure data integrity:

```
server {
    listen 443 ssl;
    server_name api.linkwave.org;
    ssl_certificate /etc/letsencrypt/live/api.linkwave.org/
       fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/api.linkwave.org/
       privkey.pem;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-
       SHA256:...;
}
```

Security Headers

To protect the system against common attack vectors, the following security headers have been implemented:

```
add_header X-Frame-Options "SAMEORIGIN";
add_header X-Content-Type-Options nosniff;
add_header X-XSS-Protection "1; mode=block";
```

Proxy Configuration

Nginx acts as a proxy to enhance security by forwarding requests to services running in Docker containers on an internally accessible port.

```
location / {
    proxy_pass https://localhost:8082;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
}
```

Server Security

Two user accounts are set up on the server: ‘linkwave-admin‘ for administrative tasks and ‘linkwave-web‘ for web applications and hosting. The ‘linkwave-web‘ account has restricted privileges, adhering to the principle of least privilege, thereby increasing system security. By dividing user roles, only authorized actions can be performed, minimizing the risk of security breaches.

5 Implementation of Features

This section describes the implementation of the individual features of LinkWave across various operating systems. It also explains the specific functionality and limitations on each platform.

5.1 Device Discovery

Device discovery enables users to find other devices in their vicinity and communicate with them without having to add them manually.

Device discovery is implemented in the LinkWave app on all operating systems. It is essential for the use of all LinkWave features. To simplify the usage of LinkWave, it is crucial that device discovery operates automatically, without user interaction, and reliably.

This is also a unique selling point compared to other “connectivity apps” like KDE Connect, where devices must be added manually. This step is unnecessary in LinkWave, as device discovery runs automatically.

5.1.1 Functionality

The following figure illustrates the use of Bonjour or mDNS by LinkWave to enable the discovery and communication with other devices on the network.

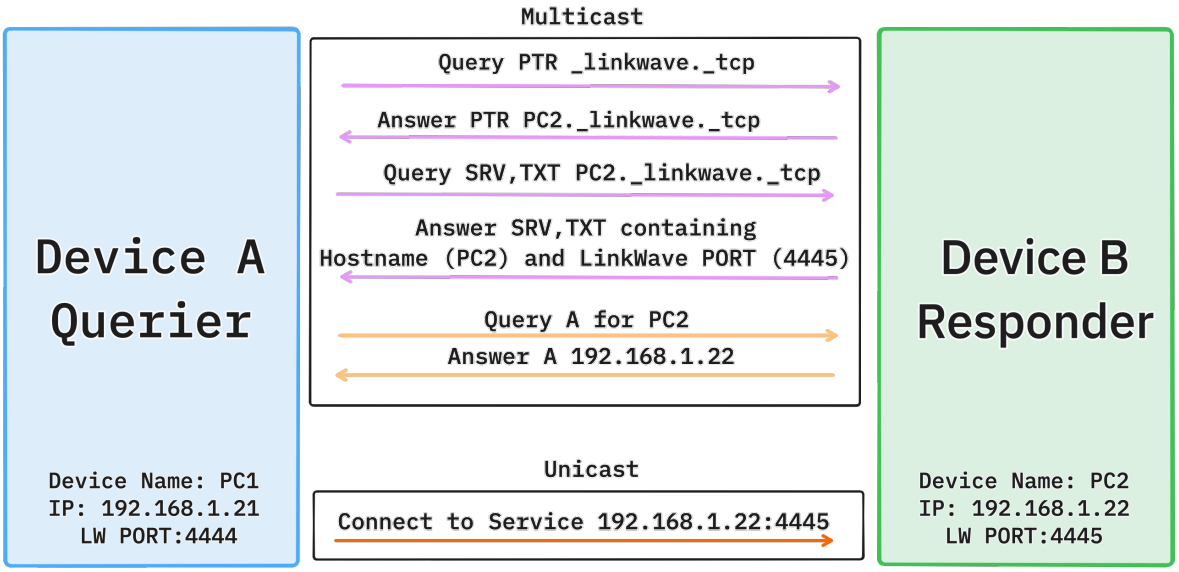


Figure 27: Device Discovery

1. **mDNS Query:** A device (Device A) sends an mDNS query in multicast mode to discover services using “_linkwave._tcp”. This query is visible to all devices on the network.

2. **mDNS Response:** Another device (Device B) responds with a PTR (Pointer) Record response, which contains the name of the service (“PC2._linkwave._tcp”).

3. **Service and TXT Records:** Device A then queries for SRV (Service) and TXT (Text) Records to obtain the hostname (PC2), the port of the LinkWave service (4445), and additional information such as the public key.

4. **A-Record Query:** To find the IP address of the service, Device A sends an A-Record query for “PC2”.

5. **A-Record Response:** Device B responds with the IP address “192.168.1.22”.

6. **Connection Establishment:** Finally, Device A establishes a unicast connection to the LinkWave service by connecting to the IP address and specified port (“192.168.1.22:4445
   The port is dynamically chosen based on the available ports in the operating system used for the LinkWave TCP listener.

5.1.2 On iOS and macOS

Bonjour or zero-configuration networking is implemented in iOS and macOS together and is provided via Apple's `NWBrowser` and `NWListener` classes. As the creator of Bonjour, Apple offers classes for easy usage. The device discovery and advertising of the local device in LinkWave are implemented by the `LinkWaveHelper` class.

Device Discovery Device discovery using Bonjour can be implemented very easily in Swift through the `NWBrowser` class. First, an instance of the `NWBrowser` class is created and configured. The configuration includes the service type, protocol, and domain. How LinkWave uses Bonjour in detail is described in Chapter 5.1.2.

```
let browser: NWBrowser = .init(
  for: .bonjourWithTXTRecord(
  type: "_linkwave._tcp",
    domain: "local"
  ),
  using: .tcp
)
browser.browseResultsChangedHandler = ResultsChangedEventHandler
```

To use device discovery, an event handler is required to respond to new or removed devices. This is defined through the function `ResultsChangedHandler`. In this function, a change set is received, which contains the added and removed devices. Each change is then processed in a loop. The change is a selection of the `NWBrowser.Result.Change` enum. In the loop, the endpoint of the device can be extracted from the change, and the device can be added or removed accordingly. This is possible because enums in Swift can have associated values for different cases.

```
private func ResultsChangedEventHandler(
results _: Set<NWBrowser.Result>,
changes: Set<NWBrowser.Result.Change>
) {
for change in changes {
switch change {
  case let .added(endpoint):
  // Add device to the list of visible devices
  case let .removed(endpoint):
  // Remove device as it is no longer visible
}
}
}
```

From this endpoint, additional information such as the device's name or IP address can be extracted. LinkWave also uses a TXT record to transmit additional information like the device type. TXT records are stored in the endpoint under `endpoint.metadata.dictionary`

After these queries, the endpoint with the additional information is wrapped into a `LinkWaveDevice` object and added to a list or removed from it. This list is then passed to the UI to display the devices.

The `LinkWaveDevice` object contains the following information:

- **Name:** A unique identifier for each device.
- **Type:** Describes the type of device (e.g., Windows, Mac, Android).
- **Origin:** The origin of the device. This value is an enum that can take the values "Network" and "Bluetooth." The value "Network" means the device was discovered via the network. The value "Bluetooth" is intended for future extensions. In the enum value "Network," the endpoint of the device is also stored.

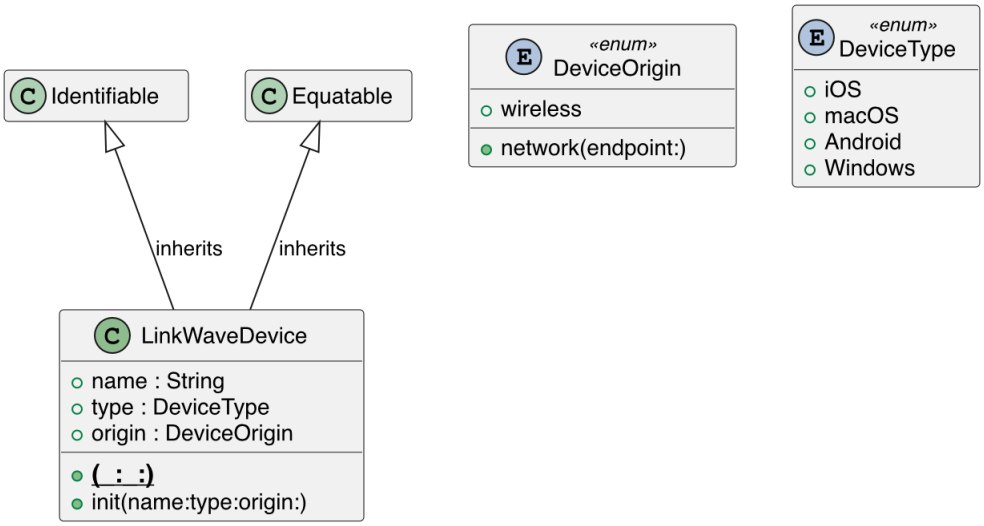


Figure 28: UML of LinkWaveDevice

Advertising the Local Device Advertising the local device in LinkWave is also implemented using Apple's `NWListener` class. The `NWListener` is initialized with a service type and a random port. The service type is the same as in device discovery since all LinkWave devices use the same service type.

```
let listener: NWListener = try! NWListener(using: .tcp, on: .any)
listener.service = .init(
  name: deviceInfo.deviceName,
  type: "_linkwave._tcp",
  domain: "local"
)
```

The `NWListener` is then configured with a TXT record that contains additional information, such as the device type. Using the compiler directive `#if os(iOS)`, the device type is set to either iOS or macOS. Based on the operating system, the variable `osCode` is assigned the value "iOS" or "macOS." The TXT record is passed as a dictionary.

```
#if os(iOS)
  let osCode = "iOS"
#elseif os(macOS)
  let osCode = "macOS"
#endif
let txtRecord = ["deviceType": osCode]
listener.service.txtRecord = .init(txtRecord)
listener.newConnectionHandler = NewConnectionHandler
```

The `NWListener` is assigned a `NewConnectionHandler`, which responds to new connections. In this handler, the connection is accepted, and the various functions of LinkWave are invoked. It is essential that this handler runs on a background thread to prevent blocking the UI.

```
private func NewConnectionHandler(newConnection: NWConnection) {
  DispatchQueue.global(qos: .background).async {
    newConnection.stateUpdateHandler = { newState in
    switch newState {
      case .ready:
        newConnection.receive() { content, _, _, _ in
          let linkWaveCommandRequest = JSONDecoder()
            .decode(LinkWaveCommandRequest.self, from: content!)
          swift linkWaveCommandRequest.RequestType {
            case .sendFile: // Receive file
            // Other requests
} } } } }
```

Incoming connections send a JSON object, which is decoded into a `LinkWaveCommandRequest` object. This object contains information about the request type. Based on the content of the request, the corresponding function is then called.

5.1.3 On Android

Bonjour or zero-configuration networking is implemented in Android as the Network Service Discovery Protocol (NSD) and is provided through the `NsdManager` class. Device discovery and advertising the local device in the LinkWave app are implemented in two classes: `BonjourController` and `BonjourAdvertiser`.

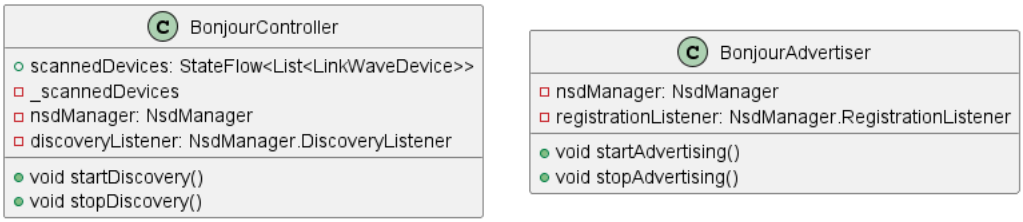


Figure 29: UML of BonjourController and BonjourAdvertiser

BonjourController

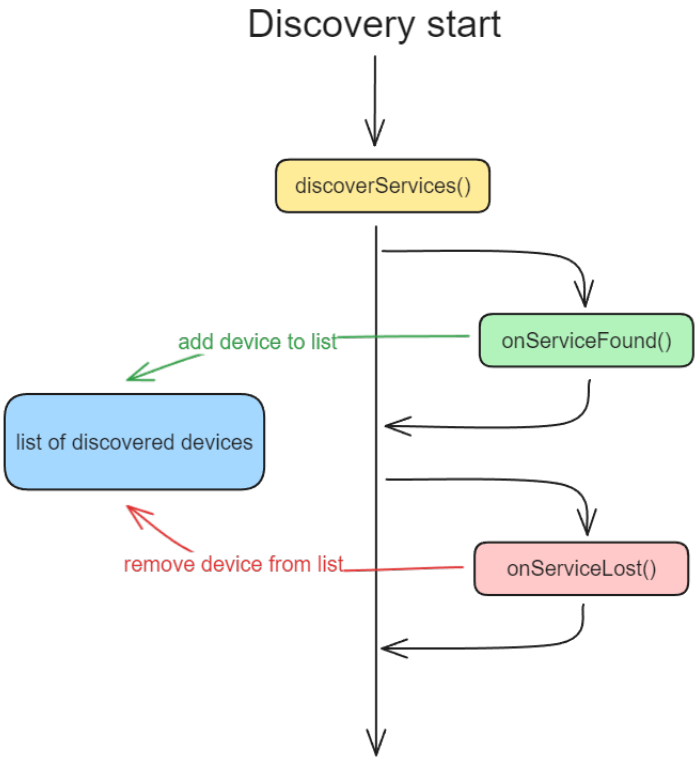


Figure 30: Process of Device Discovery

The `BonjourController` is responsible for scanning for devices on the local network and is available in the app's ViewModels via dependency injection. Figure 30 shows the device discovery process. The `discoverServices()` method provided by the NsdManager starts the discovery process. A `discoveryListener` is passed to it, which provides callback methods for the discovery process, allowing the detected devices to be processed.

Initialization The instance of the `NsdManager` is initialized using the app's `androidContext`. The `androidContext` contains information about the "application environment." In this context, it is used to access the system service NSD. The following code snippet shows the initialization of the NsdManager.

```
private val nsdManager by lazy {
    androidContext
        .getSystemService(Context.NSD_SERVICE) as NsdManager
}
```

discoverServices() The `BonjourController` provides the `startDiscovery()` method, which starts the device discovery process. This method calls the `discoverServices()` method of the `NsdManager` to initiate the discovery. The service type, protocol, and `discoveryListener` are passed to it.

```
fun startDiscovery() {
    [...]

    nsdManager.discoverServices(
        BonjourConfig.SERVICE_TYPE,
        NsdManager.PROTOCOL_DNS_SD,
        discoveryListener
    )
}
```

- **SERVICE_TYPE:** The service type is the service identifier used to discover devices. Similar to the hostname in traditional DNS, it does not identify a single host but rather all hosts offering a specific service. All LinkWave devices use the same service type: `_linkwave._tcp`.
- **PROTOCOL_DNS_SD:** The protocol used for device discovery. Here, the DNS Service Discovery protocol is used.
- **discoveryListener:** The implementation of the `NsdManager.DiscoveryListener` interface. This interface provides callback methods for the discovery process: when the search for services starts, when a service is found or lost, and when an error occurs. "Lost" in this context means a service is no longer available. Figure 31 shows the methods of the `DiscoveryListener`.

Figure 31: UML of DiscoveryListener

onServiceFound() When a device is found, the `onServiceFound()` method of the `discoveryListener` is called. Now, connection information such as the IP address and port number must be retrieved using the `nsdManager.resolveService()` method. This method also receives an object with callback methods that are triggered when the information is retrieved or an error occurs. Since Android 14, an implementation of the `NsdManager.ServiceInfoCallback` interface can be used for this purpose. Unlike the outdated `NsdManager.ResolveListener`, the `ServiceInfoCallback` can respond when an attribute of a service changes. For instance, if the IP address of a device changes, the ServiceInfoCallback can handle this. However, as this interface is only available starting with Android 14, LinkWave still uses the `ResolveListener`. The `ResolveListener`

receives an `onServiceResolved()` callback, which is called when a specific service is resolved. If an attribute of a device changes, it must be resolved again since the `ResolveListener` is discarded after the first resolution and cannot respond to changes of the resolved service.

```
override fun onServiceFound(serviceInfo: NsdServiceInfo?) {
    [...]
    val resolveListener = ResolveListener { resolvedServiceInfo ->
        _scannedDevices.update { devices ->
            val newDevice = BonjourDevice
                .fromNsdServiceInfo(resolvedServiceInfo)
            if (newDevice in devices) devices
            else devices + newDevice
        }
    }

    nsdManager.resolveService(serviceInfo, resolveListener)
}
```

onServiceLost() When a device is lost, the `onServiceLost()` method of the `discoveryListener` is called. In this method, the device is removed from the list of discovered devices.

```
override fun onServiceLost(serviceInfo: NsdServiceInfo?) {
    [...]
    _scannedDevices.update { devices ->
        devices.filter { it.name != serviceInfo.serviceName
            && it.address != serviceInfo.host }
    }
}
```

Updating the User Interface The list of discovered devices is provided via a `StateFlow` in the ViewModels of the app. The `BonjourController` updates this `StateFlow` with the discovered devices. The ViewModels of the app can subscribe to this `StateFlow` and always receive the current list of discovered devices. Kotlin Flow is a library that en-

ables the processing of asynchronous data streams. The following code shows how the ViewModels are used in the app's Activities to display the discovered devices.

```
class ShareActivity() : ComponentActivity() {
    private val devicesViewModel: DevicesViewModel by viewModel()

    setContent {
        val devicesState = bonjourViewModel.state
            .collectAsState()

        DeviceList(
            devices = bonjourState.value.scannedDevices,
        )
    }
}
```

BonjourAdvertiser

Figure 32: Process of Advertising the Device

The `BonjourAdvertiser` is responsible for advertising the device on the local network. It is managed by the `LinkWaveService`, which is described in more detail in Chapter 5.5.3. When the service starts, the `BonjourAdvertiser` is initialized, and the device is advertised on the local network.

startAdvertising() The `BonjourAdvertiser` is initialized when the `LinkWaveService` starts. For this purpose, the `startAdvertising()` method is called, which registers a service with the name of the device using the `NsdManager`. The device information is stored in an `NsdServiceInfo` object and passed to the `NsdManager`.

```
val serviceInfo = NsdServiceInfo().apply {
    serviceName = BonjourConfig.serviceName
    serviceType = BonjourConfig.SERVICE_TYPE
    port = BonjourConfig.SERVICE_PORT
    setAttribute("deviceType", LocalDeviceInfo.type.name)
}
```

- **serviceName:** The name of the service that will appear on the local network. Here, the device name is used.
- **serviceType:** The service type that will appear on the local network. Here, LinkWave's service type is used: `_linkwave._tcp`.
- **port:** The port on which the service is accessible. LinkWave uses a random port.

- **setAttribute:** Additional information about the device can be stored here. In this case, the device type is stored.

The process of advertising the device on the network is started with the `registerService` method of the `NsdManager`. This method receives the `NsdServiceInfo` object and a `registrationListener`. The `registrationListener` provides callback methods that are called when the process is successful or when an error occurs.

Figure 33: UML of RegistrationListener

```
suspend fun startAdvertising() = withContext(Dispatchers.IO) {
    val serviceInfo = [...]

    nsdManager.registerService(
    serviceInfo,
    NsdManager.PROTOCOL_DNS_SD,
    registrationListener
)
}
```

stopAdvertising() The `BonjourAdvertiser` provides the method `stopAdvertising()` to remove the local device from the network. For this, the `unregisterService()` method of the `NsdManager` is called.

```
suspend fun stopAdvertising() = withContext(Dispatchers.IO) {
nsdManager.unregisterService(registrationListener)
}
```

5.1.4 On Windows

For device discovery on Windows, a library called “Zeroconf” is used. The “Zeroconf” library allows sending mDNS queries and receiving mDNS responses. Device discovery is implemented via the `Discovery` and `Advertise` classes. The methods of these classes are used or called within the `LinkWaveService` class in LinkWaveUI.Core. The `Discovery` class handles scanning for devices on the local network, while the `Advertise` class is responsible for advertising the local device.

Discovery

The `FindLinkWaveDevices()` function uses the ZeroconfResolver library to discover devices that offer LinkWave services on the local network (other devices with LinkWave installed). To achieve this, a listener is created that searches for mDNS (Multicast DNS) entries published under the service name “_linkwave._tcp.local”.

Figure 34: UML Discovery

```
public static class Discovery
{
public static ZeroconfResolver.ResolverListener FindLinkWaveDevices
    ()
{
     var options = ZeroconfResolver
     .CreateListener("_linkwave._tcp.local.");
     return options;
}
}
```

Advertise

The method RegisterService() uses the service name “_linkwave._tcp” to create a service description, which is then announced on the network. The ServiceProfile instance is configured with the given instance name, the service name, and the port. Subsequently, this configuration is announced on the local network using the Advertise method, allowing other devices to discover and interact with the service.

Figure 35: UML Advertisement

```
public class Advertisement
{
    public static async Task RegisterService(string instance,
        ushort port)
    {
        string serviceName = "_linkwave._tcp";
        var service = new ServiceProfile(instance, serviceName,
            port);
        // Set service fields
        sd.Advertise(service);
    }
}
```

5.1.5 On the Server

The following section describes the GraphQL endpoints for device discovery on the server:

Database Structure

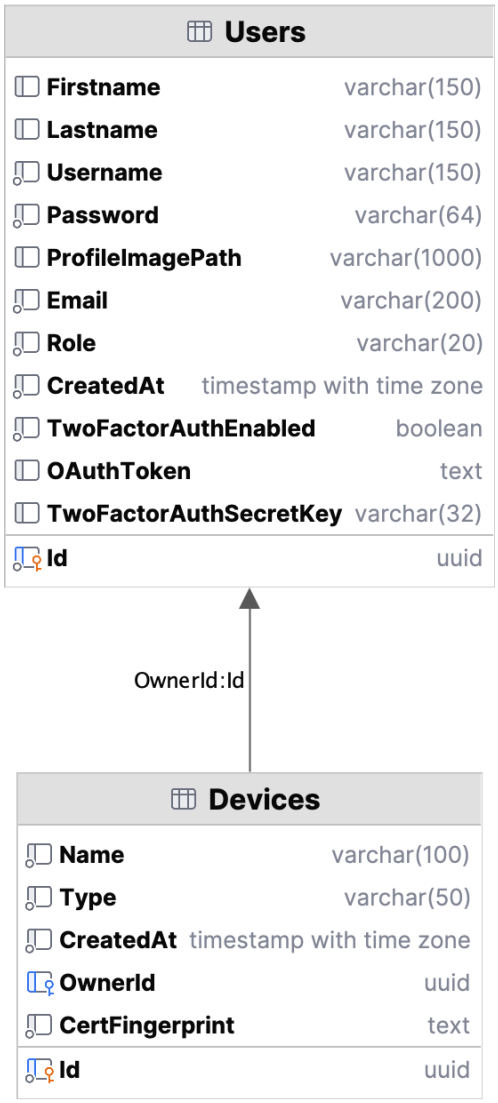


Figure 36: Devices Table

The table consists of several fields that work together to store comprehensive data for each device:

- **Id (UUID):** Unique identifier for each device.
- **Name (String):** Name of the device, provided by the user for easy identification.
- **Type (String):** Describes the type of the device (e.g., Windows, Mac, Android).
- **Owner (User):** References the owner of the device in the 'Users' table.
- **CertFingerprint (String):** A text field that stores the certificate fingerprint generated when an account is registered on a device.
- **CreatedAt (DateTime):** The date and time the device entry was created.

GraphQL Endpoints for Devices

AddDevice The GraphQL mutation endpoint `AddDevice` allows users to add new devices to the system. The endpoint expects device information and stores it in the 'Devices' table. The associated user is updated to reflect the new device in their device list.

```
public async Task<Database.Models.Device> AddDevice(LinkWaveContext
    dbContext, Guid userId , string deviceName, string deviceType,
   string fingerprint)
{
    ...
    dbContext.Devices.Add(device);
    user.Devices.Add(device);
    return device;
    // Error handling is omitted
}
```

GetDevicesByUserId The query endpoint `GetDevicesByUserId` allows users to retrieve a list of all registered devices associated with their account. This endpoint returns

a collection of device data that can be used for management or verification purposes by the user.

```
public async Task<List<Database.Models.Device>> GetDevicesByUserId(
    LinkWaveContext dbContext, Guid userId)
{
    var usersDevices = await dbContext.Users.SelectMany(x => x.
        Devices).Where(x => x.Owner.Id == userId).ToListAsync();
    return usersDevices;
}
```

UpdateDeviceDetails The `UpdateDeviceDetails` endpoint is used to update the data of an existing device. This can be required for various scenarios, such as updating the certificate fingerprint or device name, ensuring the device data is up-to-date and maintaining system integrity.

```
public async Task<Database.Models.Device> UpdateDeviceDetails(
    LinkWaveContext dbContext, Guid deviceId, string newDeviceName,
    string newCertFingerprint)
{
    ...
    deviceToUpdate.CertFingerprint = newCertFingerprint;
    dbContext.Devices.Update(deviceToUpdate);
    return deviceToUpdate;
}
```

DeleteDevice The `DeleteDevice` endpoint is responsible for removing devices from the system.

```
public async Task<bool> DeleteDevice(LinkWaveContext dbContext,
    Guid deviceId)
{
     ...
     dbContext.Devices.Remove(deviceToRemove);
     await dbContext.SaveChangesAsync();
     return true;
}
```

5.2 File Transfer

File transfer is one of the most important features of LinkWave. It enables files to be transferred between different devices. File transfer works without restrictions on all operating systems.

To make the experience as seamless as possible for users, a uniform workflow was designed. The recommended workflow for file transfer is as follows:

1. **Share:** When using other apps, you need to use the system-specific sharing function for the files you want to transfer.
2. **Select LinkWave:** In the menu that opens, you can select LinkWave. Once LinkWave is installed, it automatically integrates into this menu.
3. **Select Device:** You then need to select the device to which the files should be sent. This device must be nearby since the file transfer occurs via a direct connection between the devices.
4. **Confirm:** Once the device is selected, the target device receives a notification. In this notification, you can confirm or reject the file transfer. Once confirmed, the file transfer begins. The file appears immediately in the location selected in the settings.

These four steps provide the same workflow as Apple's "Airdrop," which is highly regarded for its simplicity and serves as a benchmark for LinkWave.

5.2.1 Functionality

The file-sharing feature of LinkWave is designed as a secure and user-friendly process based on an automated and encrypted communication flow. The functionality of LinkWave's file-sharing feature is described below as represented in the application's flow diagram:

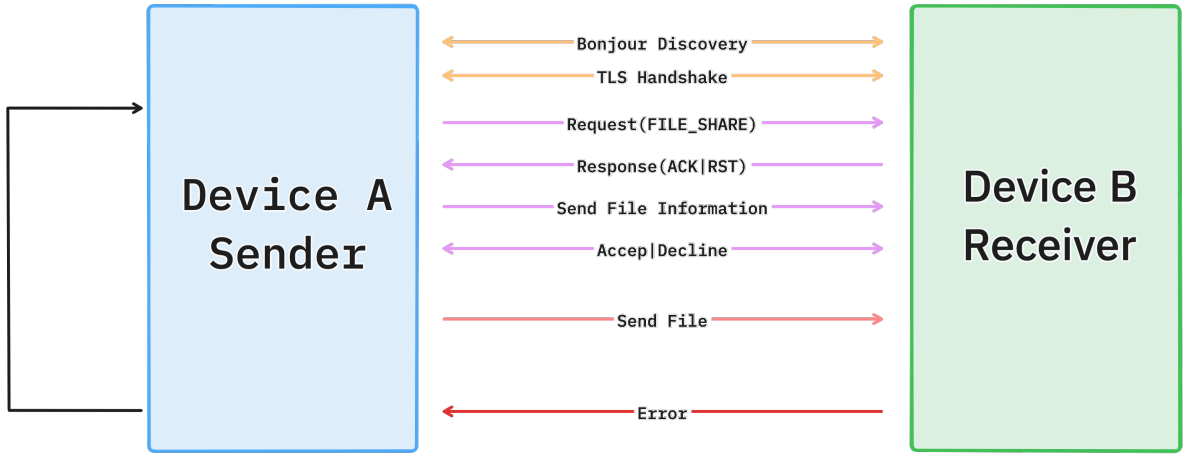


Figure 37: File Sharing

1. **Bonjour Discovery:** The process begins with the discovery of devices with LinkWave installed via the Bonjour protocol. LinkWave uses the Bonjour protocol to automatically detect devices on the local network.
2. **TLS Handshake:** After device discovery and recipient selection, LinkWave establishes a secure connection using TLS (Transport Layer Security), represented by the TLS handshake. This ensures encrypted communication between devices and protects transferred files from potential security risks.
3. **Request (FILE_SHARE):** The sending device (Device A) signals its readiness to exchange data through a corresponding request to the receiving device (Device B), initiating the data exchange.
4. **Response (ACK|RST):** Device B responds to the request either with an acknowledgment (ACK) or a reset (RST). An ACK continues the process, while an RST rejects the request and terminates the process.

5. **Send File Information:** After successful acknowledgment, Device A sends information about the file to be shared to Device B. LinkWave uses this information to provide the recipient with details such as file size, type, and name.

6. **Accept/Decline:** Based on the received file information, Device B can choose to accept or decline the file. LinkWave provides a user interface for the recipient to make this decision.

7. **Send File:** If the recipient accepts the file, Device A begins transmitting the file to Device B. LinkWave manages the transfer process and displays the transfer progress.

8. **Error:** If errors occur during the transfer, LinkWave provides mechanisms for error detection and reporting.

5.2.2 On iOS and macOS

File transfer is implemented on iOS and macOS using TCP. The file transfer is realized via Apple's `NWConnection` class. The code is identical on both operating systems since the `NWConnection` class is available in Swift on both platforms.

Sending Files

To send files, the `sendFileHandler` function can be used. This function takes the following parameters:

- **Endpoint (NWEndpoint):** The endpoint of the device to which the file should be sent.
- **FileStream (InputStream):** The stream of the file to be sent.
- **FileName (String):** The name of the file to be sent.
- **FileSize (Int64):** The size of the file to be sent.
- **FileSendProgress (inout Double):** An inout parameter through which the progress in percentage is forwarded to the UI.
- **AcceptedCallback ((Bool) -> Void):** A callback function called when the file transfer is accepted or declined.
- **CompletionCallback ((Bool) -> Void):** A callback function called when the file transfer is completed.

In this function, the file transfer is carried out as described in Chapter 5.2.1. The file stream is split into small packets and sent sequentially to the device. The progress of the file transfer is forwarded to the UI so that the user can see the progress. Once the file transfer is complete, the `CompletionCallback` function is called.

The `send()` and `receive()` methods provided by Apple in the `NWConnection` class are asynchronous and do not block the main thread. These methods handle the transfer of all data packets. To create an `NWConnection` from a Bonjour endpoint, the endpoint is simply passed to the constructor. The `NWConnection` can then be started on the main

thread using the `start(queue: .main)` method. Data packets are transmitted asynchronously. The `send()` and `receive()` methods work with callbacks that are invoked when data is sent or received.

```
connection.receive(
    minimumIncompleteLength: 1, maximumLength: 1024
) { content, _, _, error in
    // Process data
}
```

The file is read from an `InputStream` in chunks and sent to the device using the `send()` method. The `InputStream` is read in chunks of 1024 bytes and transmitted to the device. The progress of the file transfer is forwarded to the UI so the user can monitor it. This process is carried out in a loop until the entire file is transferred. Thanks to the callback mechanism, there is no need to wait for the transfer of one chunk to complete before sending the next, thus speeding up the transfer process.

```
var buffer = [UInt8](repeating: 0, count: transferChunkSize)
while case let amount = fileStream.read(
    &buffer,
    maxLength: transferChunkSize),
    amount > 0 {
    let chunk = Array(buffer[..<amount])
    connection.send(content: chunk) { error in
    // Update UI upon successful transfer
    }
}
```

Receiving Files

Receiving files in the LinkWave app is implemented by the `LinkWaveHelper` class. This is another function that can be invoked in the `NewConnectionHandler` (see Chapter 5.1.2).

Once the connection is established, the information about the file to be received is transmitted. This information includes the file name and size. A popup is then displayed,

allowing the user to accept or reject the file transfer. Additionally, a callback is registered, which is triggered when the file transfer is accepted or rejected. If the user accepts the file transfer, the file is received and stored. The progress of the file transfer is forwarded to the UI so the user can monitor it.

5.2.3 On Android

The following section describes the implementation of file transfer in the Android app. File transfer is implemented via a TCP connection.

Sending Files

Figure 38 shows the file transfer process on Android.

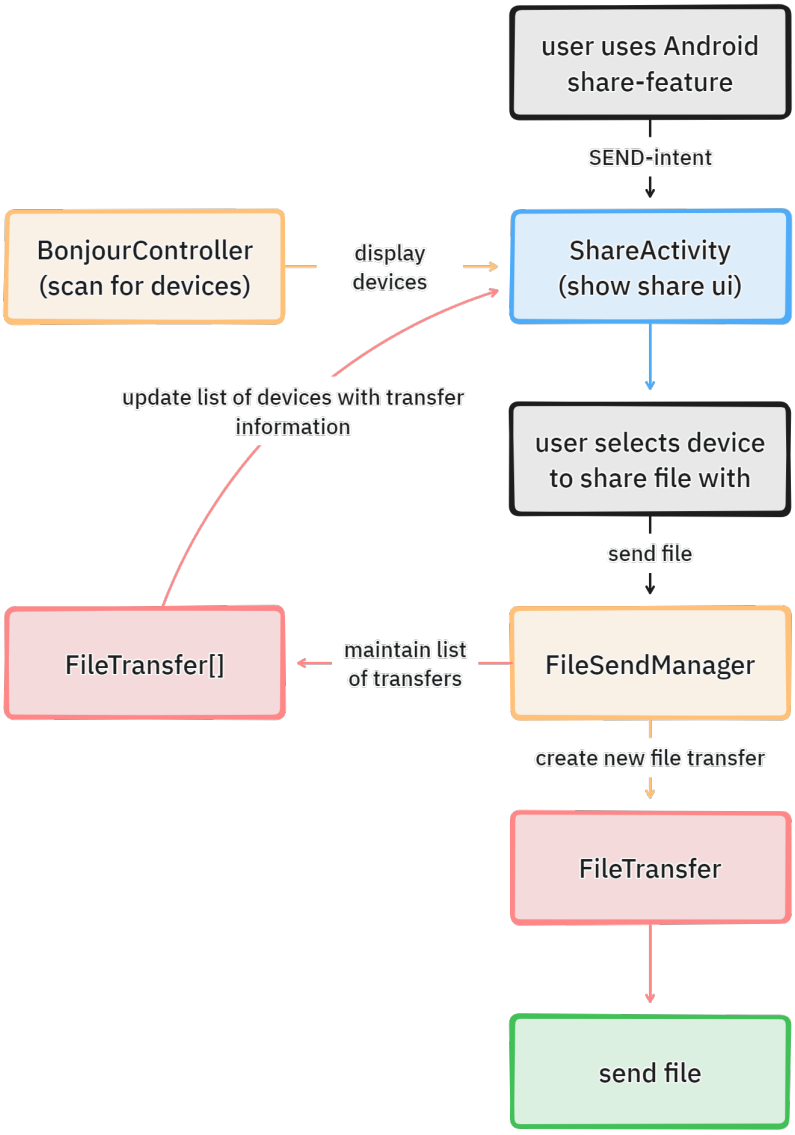


Figure 38: File Transfer Process on Android

To share a file, users must use Android's "Share" feature. When the LinkWave app is selected in the "Share" window, Android sends a `SEND` intent with a URI pointing to the file to the app. To enable the app to receive a `SEND` intent, an `intent-filter` must be configured. This is defined in the `AndroidManifest.xml` file.

```
<activity
    android:name=".ui.share.ShareActivity"
    android:excludeFromRecents="true"
    android:exported="true"
    android:launchMode="singleInstance"
    android:theme="@style/Theme.App">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="*/*" />
    </intent-filter>
</activity>
```

Here is the `intent-filter` that intercepts the `SEND` intent. The `intent-filter` is configured to receive all file types. The `mimeType` specifies the type of file. The `mimeType` for a `jpeg` image is `image/jpeg`. `*/*` indicates that the filter applies to all file types.

ShareActivity The `intent-filter` belongs to the `ShareActivity`. When the app receives a `SEND` intent, either the `onCreate()` method or the `onNewIntent()` method is called. Which method is invoked depends on the `launchMode` and whether the Activity is already running in the background. `singleInstance` ensures that only one instance of the Activity exists. If the Activity is already in the background, the `onNewIntent()` method is called; otherwise, the `onCreate()` method is called.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    intent?.let { handleIntent(it) }

    // ... Initialize UI
}

override fun onNewIntent(intent: Intent?) {
    super.onNewIntent(intent)
    intent?.let { handleIntent(it) }
}
```

handleIntent() In the `handleIntent()` method, the intent is processed. The URI contained in the intent is passed to the `ShareViewModel`, which reads the file's data and temporarily stores it as `selectedFile`.

```
private fun handleIntent(intent: Intent) {
    when (intent.action) {
        Intent.ACTION_SEND -> {
            val uri = intent.getParcelableExtra(Intent.EXTRA_STREAM
                , Uri::class.java)
            if (uri == null) return

            shareViewModel.selectFile(uri, this)
        }
        [...]
    }
}
```

The selected file is displayed in the `ShareActivity`. Now, a device from the `DeviceList` must be selected to which the file should be sent. The following code snippet shows how the devices and the selected file are displayed in the `ShareActivity`.

```
override fun onCreate(savedInstanceState: Bundle?) {
    setContent {
```

```
        val fileState = shareViewModel.state.collectAsState()
        val devicesState = devicesViewModel.state.collectAsState()

        FileArea(file = fileState.value.selectedFile)

        DeviceList(
            devices = devicesState.value.scannedDevices,
            fileTransfers = fileState.value.fileTransfers,
            onSelect = { device ->
                shareViewModel.sendFile(device)
            }
        )
    }
}
```

FileSendManager The `FileSendManager` manages outgoing file transfers. It is used by the `ShareViewModel` to send files. Figure 39 shows the class diagram of the `FileSendManager`.

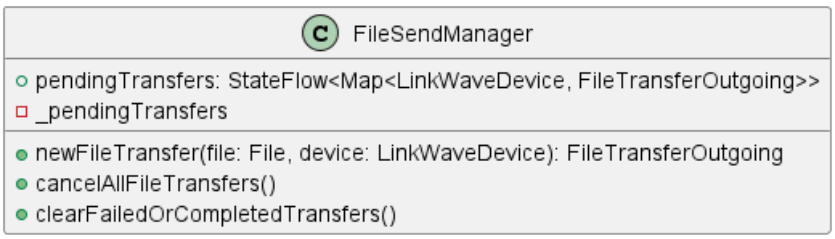


Figure 39: UML of FileSendManager

The `FileSendManager` stores the current file transfer in a HashMap. The `newFileTransfer()` method creates a new file transfer and saves it in the HashMap. The `send()` method of the `FileTransfer` is called to start the file transfer.

```
fun newFileTransfer(file: AndroidFile, device: LinkWaveDevice) {
    if (_pendingTransfers.value.containsKey(device)) {
        return
    }
```

```
    val fileTransfer = FileTransferOutgoing(androidContext, file,
        device)

    _pendingTransfers.update { transfers ->
    transfers + (device to fileTransfer)
}

fileSendManagerScope.launch {
    fileTransfer.send()
}
```

FileTransferOutgoing A `FileTransferOutgoing` object is responsible for sending a file to another device. It uses a `TCP-Socket` connection to send the file and provides information about the status and progress of the file transfer.

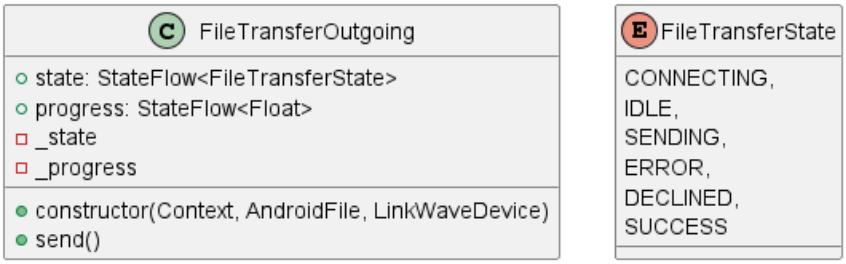


Figure 40: UML of FileTransferOutgoing

send() The exact process of file transfer is described in Section 5.2.1. The `Request` objects as well as the file information are serialized as `JSON` and sent to the receiving device via the `Socket` connection. The following code snippet demonstrates sending the file information.

```
val outputStream = socket.getOutputStream()

val fileInfo = FileShareRequest(
    file.name,
```

```
    file.size,
    file.calculateChecksum(androidContext)
)

outputStream.writeJson(fileInfo)
```

The `writeJson` extension function serializes any object as `JSON` and writes the serialized data to the `OutputStream`. The `kotlinx.serialization` library [7] is used for this. To receive the response to the `FileShareRequest` query, the `readJson` extension function is used.

```
val fileShareResponse =
    inputStream.readJson<FileShareResponse>(1024)
```

If the request is accepted, the file is read in chunks and sent to the receiving device via the `Socket` connection. The progress of the file transfer is forwarded to the UI so that the user can monitor it. The following code snippet shows the process of sending the file.

```
while (stream.read(buffer, 0, buffer.size)
    .also { read -> bytesRead = read } != -1
) {
    outputStream.write(buffer, 0, bytesRead)
    bytesTotalRead += bytesRead
    _progress.value = bytesTotalRead.toFloat() / file.size
}
```

Receiving Files

Since receiving files should also work in the background, even when the app is closed, the receiving process runs within the `LinkWaveService` (see Section 5.5.3). The receiving process is illustrated in Figure 41.

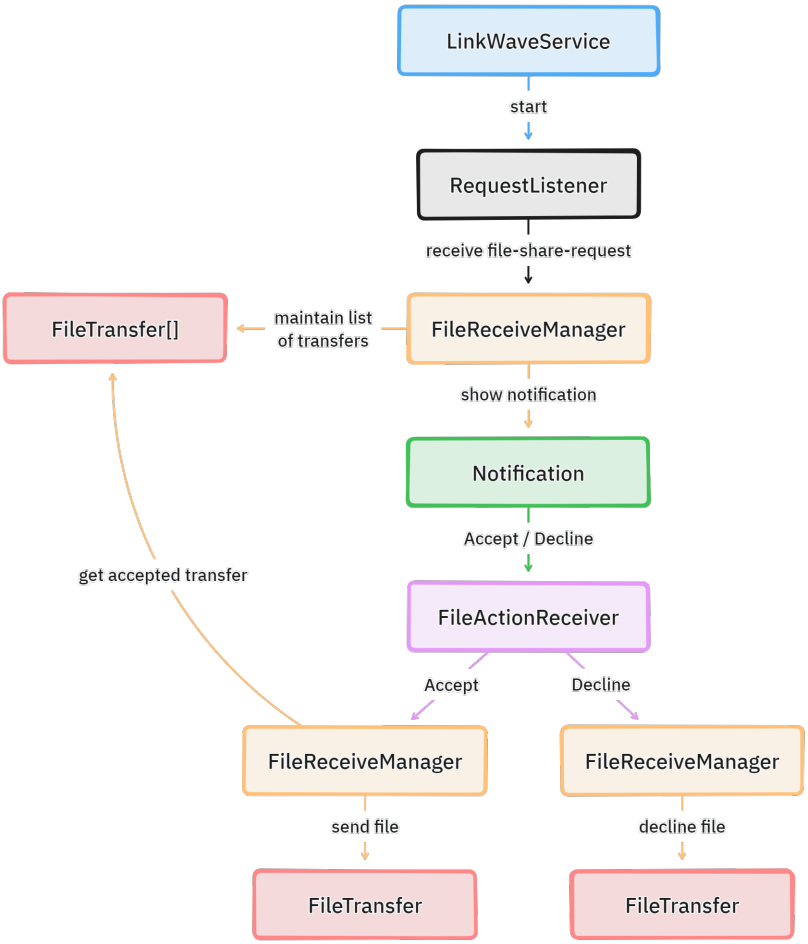


Figure 41: File Transfer Process on Android

RequestListener The `RequestListener` is launched by the `LinkWaveService` in its own coroutine. It is responsible for processing incoming requests. Figure 42 shows the class diagram of the `RequestListener`.

Figure 42: UML of RequestListener

FileReceiveManager When a `FILE_SHARE` request is received, it is forwarded to the `FileReceiveManager`. The `FileReceiveManager` stores the current file transfers in a HashMap and displays a notification to the user, allowing them to accept or decline the file transfer.

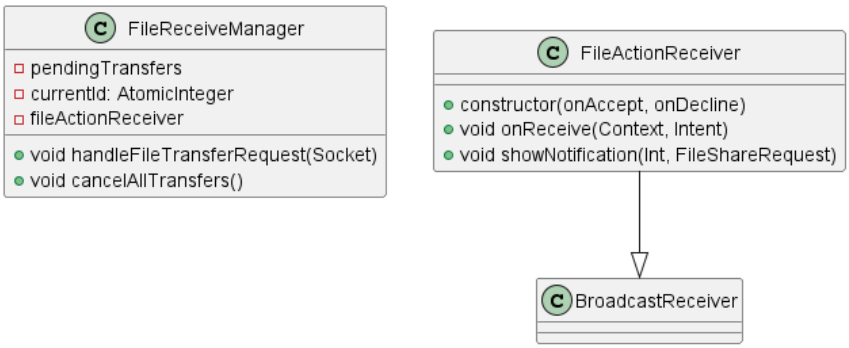


Figure 43: UML of FileReceiveManager and FileActionReceiver

FileActionReceiver The `FileActionReceiver` is a `BroadcastReceiver` that responds to the acceptance or rejection of a file transfer. In the notification shown to the user, two actions are defined, which, when selected, send the respective intent to the `FileActionReceiver` to either accept or decline the file.

```
override fun onReceive(context: Context?, intent: Intent?) {
    // read id from intent...
    when (intent.action) {
    ACTION_FILE_ACCEPT -> {
        onAccepted(id)
    }

    ACTION_FILE_DECLINE -> {
        notificationManager.cancel(id)
        onDenied(id)
    }
    }
}
```

The `onAccepted()` and `onDenied()` callback methods are defined in the `FileReceiveManager`. The `onAccepted()` function is called when the user accepts the file transfer. In this function, the file transfer is started. The `onDenied()` function is called when the user declines the file transfer. In this function, the file transfer is canceled. The following code snippet shows the `onAccepted()` function as an example.

```
private val fileActionReceiver = FileActionReceiver(
onAccepted = { id ->
    val fileTransfer = pendingTransfers[id]
        ?: return@FileActionReceiver
    fileReceiveManagerScope.launch {
        fileTransfer.receiveFile(id)
        pendingTransfers.remove(id)
    }
},
// ...
)
```

receiveFile() As with sending the file, the file transfer is received in chunks. The progress of the file transfer is displayed in the notification. The file is received in chunks

and saved.

5.2.4 On Windows

Figure 44: UML LinkWaveSender

After selecting the files to share and the recipient, LinkWave initiates the transfer process by sending a specific “Command” to the recipient device. This process is handled by the `LinkWaveSender`.

Sending Commands The `LinkWaveSender` uses an SSL network stream to send the command to the recipient.

```
public static async Task SendCommandAsync(...)
{
    // Serialization and sending of the command
    LinkWaveCommandRequest command =
    new LinkWaveCommandRequest(requestType);
    var serializedCommand = DataSerializer.SerializeObject(command)
       ;
    await stream.WriteAsync(commandBytes, 0, commandBytes.Length);
    ...
}
```

Receiving and Processing Commands

On the recipient side, the `LinkWaveReceiver` handles the incoming command. This class is responsible for waiting for incoming commands, processing them, and forwarding them to the `RequestSwitcher` class. The `StartLoopAsync` method initiates an asynchronous loop that waits for incoming connections. The method uses `SSLStream` to ensure a secure connection. `SSLStream` is a class that facilitates secure communication over SSL (Secure Sockets Layer) or TLS (Transport Layer Security). It is used to establish a secure connection between two devices. [10]

LinkWaveReceiver.cs

```
public async Task StartLoopAsync(...)
{
    while (true)
    {
        // SSL Handshake
        SslStream sslStream = await
        SSL.AcceptAndAuthenticateClientAsync(..., certificate);
        _ = HandleClient(sslStream);
    }
}
```

Once a connection is established, the `HandleClient` method processes the commands.

```
private async Task HandleClient(SslStream sslStream)
{
    var bytesRead = await sslStream.ReadAsync(commandBytes,...);
    _ = RequestSwitcher.Handle(linkWaveCommandRequest
    ,sslStream,_uIFunctionStore);
    // Error handling has been omitted
}
```

Handling Requests

The RequestSwitcher class acts as a central hub for routing incoming commands. It analyzes the type of each request and delegates it to the appropriate handlers responsible for specific processing. In this case, the `RequestSwitcher` activates the `FileSharingHandler`, which manages the actual file-sharing process.

```
public static async Task Handle(...)
{
    switch (commandRequest.RequestType)
    {
        case RequestTypes.FILE_SHARE:
        await FileSharingHandler
        .HandleAsync(networkStream,uIFunctionStore);
        break;
        // Other requests or RequestTypes
    }
    // Error handling omitted
}
```

File Sharing Handlers

The `FileReceiver` class plays a key role in the file-receiving process. It manages the acceptance, validation, and storage of files sent by another device.

ReceiveFileInformation This method reads incoming data from the network stream, which contains information about the file to be received. This information is deserialized and stored in a `FileShareRequest` object. The `FileShareRequest` object contains details such as the file name and file size.

```
public async Task<FileShareRequest> ReceiveFileInformation()
{
    // Receiving file information
    _sharedFile = DataSerializer.
    DeserializeObject<FileShareRequest>(...)!;
    return _sharedFile;
    // Error handling omitted
}
```

SendFileShareResponse After receiving the file information and displaying it to the user, this method allows the recipient to send a response indicating whether the file is accepted or declined. This decision is serialized as a `FileShareResponse` object and sent back to the sender via the network stream.

```
public async Task SendFileShareResponse(bool isAccepted)
{
    // Check whether the file was accepted or declined
    var responseObject =
    new FileShareResponse(isAccepted ? FileShareResponseType.ACCEPT
    : FileShareResponseType.DECLINE);
    await _networkStream
    .WriteAsync(responseBytes, 0, responseBytes.Length);
    // Error handling omitted
}
```

ReceiveFile This method handles the actual file reception. It reads the file bytes and writes them to the target folder. The progress is calculated as a percentage and forwarded to the user interface.

```
public async Task ReceiveFile(string fileSavePath)
{
    while (bytesLeftToRead > 0 && !cancelled)
    {
        ...
        var bytesRead = await _networkStream.ReadAsync(buffer, ...)
            ;
        if (bytesRead == 0) break;
        await fileStream.WriteAsync(buffer, 0, bytesRead);
        bytesLeftToRead -= bytesRead;
        // Update progress
    }
    // Error handling omitted
}
```

Sending Files

The FileSender class is responsible for initiating and executing the file transfer to a recipient. This class manages multiple steps of the transfer process, from transmitting the file information to the actual sending of the file content.

ReceiveAcknowledgement This method waits for an acknowledgment response from the recipient, indicating that they are ready to receive the file. It reads the response and verifies if it is a positive acknowledgment (ACK).

SendFileInformation Using the method ReceiveFileInformation, which was previously introduced, file information is received. The SendFileInformation method serves as the counterpart and sends the file information to the recipient.

In this step, the FileSender transmits details about the file to be sent, including the file name, size, and a hash value for integrity verification. This data is serialized and sent to the recipient via the network stream. This method supports both single-file and multi-file transfers.

ReceiveFileShareResponse After sending the file information, the sender waits for a response from the recipient regarding whether the file transfer was accepted. This method reads the response and verifies if the file transfer can proceed.

SendFileContent This core method is responsible for sending the actual file content. It opens the file, reads it in blocks, and sends these blocks over the network stream to the recipient. During the transfer, an optional progress indicator is updated to track the progress of the transfer.

```
public async Task SendFileContent(string filePath)
{
    // Loop to send the file data.
    while ((bytesRead =
    await fileStream.ReadAsync(buffer, 0, buffer.Length)) > 0)
    {
        await _networkStream
        .WriteAsync(buffer, 0, bytesRead);
        transferredBytes += bytesRead;
        var bytes = transferredBytes;
    }
    // Error handling omitted
}
```

5.3 Clipboard

The clipboard is another very important feature of LinkWave. It enables sharing text and images between different devices.

This feature aims to facilitate seamless switching between devices during work. For instance, one can compose a text message on a computer and then send it via a phone. Similarly, images copied on a phone can be pasted into a document on a computer.

The clipboard feature is at least partially available on all operating systems. On Windows, macOS, and Android, it is possible to copy and receive text and images. On iOS, it is possible to receive text and images, but only text—not images—can be copied. On mobile operating systems, the clipboard feature is also implemented via the share function. This means that on mobile devices, the clipboard cannot be directly accessed but instead requires using an additional button in the text selection menu.

5.3.1 Functionality

As mentioned in the file-sharing section, LinkWave uses a combination of a discovery protocol and TLS to automatically find devices and securely share data between two devices. After Device A (sender) discovers a compatible device via Bonjour Discovery and establishes a secure connection via TLS (Transport Layer Security), the following steps occur:

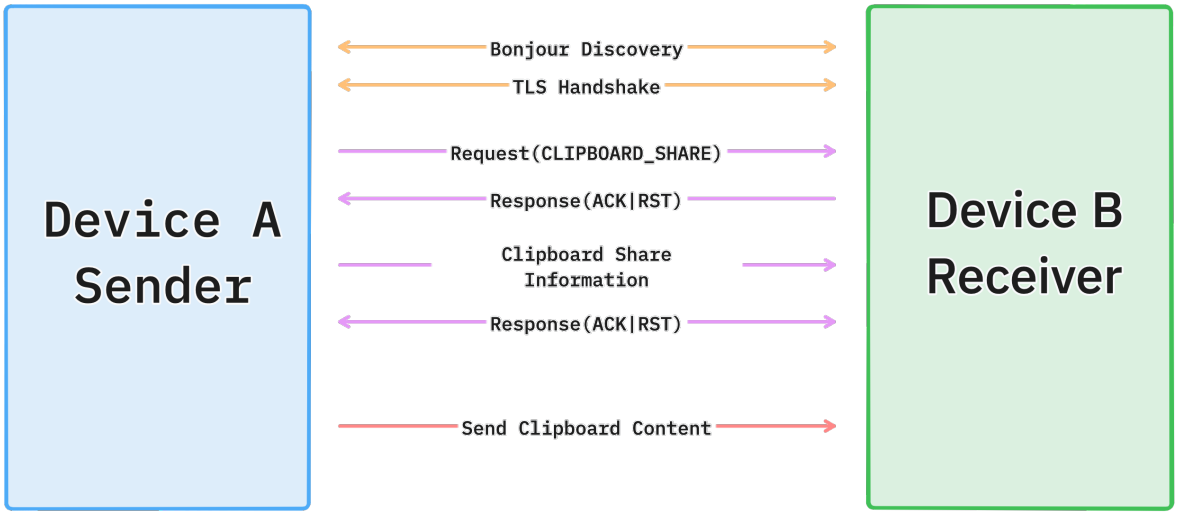


Figure 45: Clipboard Sharing

1. **Request (CLIPBOARD_SHARE):** Device A sends a request to Device B (receiver) to indicate its intention and share the clipboard content.

2. **Response (ACK|RST):** Device B responds with an ACK to confirm a successful request or with an RST to reset in case of an error.

3. **Send Clipboard Information:** Device A then sends information about the clipboard sharing. This information may include details on how the clipboard content should be formatted and transferred.

4. **Response (ACK|RST):** Device B responds to confirm that the clipboard sharing information has been received.

5. **Send Clipboard Content:** Finally, Device A sends the actual clipboard content to Device B.

5.3.2 On iOS and macOS

Since macOS and iOS share the codebase for network access, the clipboard functionality is implemented for both platforms together. Apple provides clipboard access through the `NSPasteboard` class on macOS and the `UIPasteboard` class on iOS. These classes enable copying and reading text and images to and from the clipboard.

Reading the Clipboard To read the clipboard on macOS, a timer is used to check every 2 seconds if the clipboard content has changed. The `NSPasteboard.general.changeCount` property increments with every change to the clipboard. By storing the current value in a variable and comparing it with the property's value, one can determine whether the clipboard content has changed.

```
if currentChangeCount != NSPasteboard.general.changeCount {
    currentChangeCount = NSPasteboard.general.changeCount
    // Clipboard content has changed
}
```

When the clipboard content changes, the type of the content is checked. The type is represented as an enum that stores the possible clipboard types and values. The type is then used to read the respective clipboard content.

```
let type = NSPasteboard.general.availableType(
    from: [.string, .URL, .fileURL, .png, .tiff])
switch type {
  case .string:
    // Read text from clipboard
  case .png:
    // Read image from clipboard
} // ... Other types
```

Afterward, the content is sent to the devices associated with the user's account as described in Section 5.3.1. The methods `NWConnection.send()` and `NWConnection.receive()`, described in Section 5.2.2, are used for this purpose.

Writing to the Clipboard Writing to the clipboard is another feature that can be triggered via the New Connection Handler described in Section 5.1.2. However, the SSL certificate ensures that only devices associated with the user's account can access the clipboard. Due to slight differences between macOS and iOS, two compiler directives must be used to compile the code for both operating systems.

```
#if os(macOS)
    let pasteBoard = NSPasteboard.general
    pasteBoard.clearContents()
    // ...
#elseif os(iOS)
    let pasteBoard = UIPasteboard.general
    pasteBoard.items = []
    // ...
#endif
```

In the first step, the old content must be cleared from the clipboard. Then the new content is written to the clipboard. Based on the type of the content, it is written (stored in the `data` variable) to the clipboard.

```
switch type { // on macOS
case .TEXT:
    pasteBoard.setString(
    String(data: data, encoding: .utf8),
    forType: .string
    )
case .IMAGE:
    pasteBoard.setData(data, forType: .png)
}

switch type { // on iOS
case .TEXT:
    pasteBoard.string = String(data: data, encoding: .utf8)
case .IMAGE:
    pasteBoard.image = UIImage(data: data)
}
```

The type of content is received over the TCP connection, as described in Section 5.3.1, and deserialized into the `ClipboardShareRequest` class. This class contains the type of the content and its size in bytes.

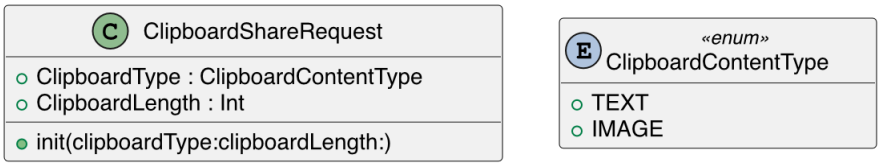


Figure 46: UML of ClipboardShareRequest

The `data` variable is an object of the same-named `Data()` class. It contains the bytes of the clipboard content. In the case of text, these are UTF-8 encoded bytes, and in the case of images, they are the image bytes. Whether it is a PNG or JPEG image does not matter, as the clipboard on macOS and iOS supports both formats and treats JPEGs as PNGs. Clipboards on other operating systems also support this ambiguity. Therefore, only the types `.TEXT` and `.IMAGE` are defined.

5.3.3 On Android

Sharing the Clipboard

Sharing the clipboard works similarly to sharing files. Instead of an image or a file, text is transferred. The process is illustrated in Figure 47.

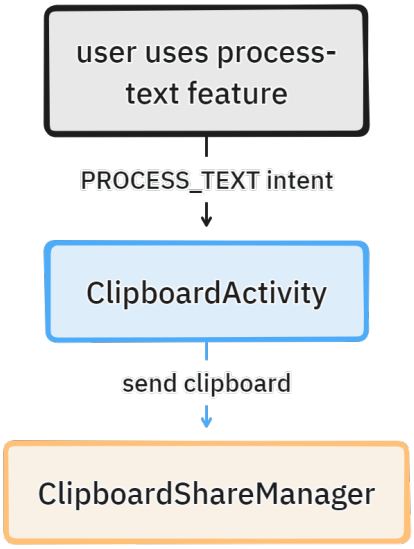


Figure 47: Clipboard Sharing Process

Since direct access to the clipboard has been restricted in Android 10 and later, the `PROCESS_TEXT` feature of Android is used. This feature allows marked text to be sent to an app, which can then process it further. In the case of the LinkWave app, a `PROCESS_TEXT` intent opens the ClipboardActivity. This activity forwards the text to the `ClipboardViewModel`, which uses the `ClipboardShareManager` to send the text to all devices.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val devicesState = devicesViewModel.state.value

    intent?.let { i ->
        val selectedText = i.getStringExtra(Intent.
            EXTRA_PROCESS_TEXT)

        clipboardViewModel.shareClipboardText(selectedText,
            devicesState.scannedDevices)
    }

    finish()
}
```

In the above code snippet, the marked text is read from the intent and forwarded to the `ClipboardViewModel`. The `finish()` function ends the activity after the text has been forwarded. This process is invisible to the user.

```
fun shareClipboardText(text: String, devices: List<LinkWaveDevice>)
    {
    for (device in devices) {
        viewModelScope.launch {
            clipboardShareManager.shareClipboardText(text, device)
        }
    }
}
```

The text is then sent as a `ByteArray` via the `NetworkStream` to the devices. The recipient can then copy the text to their clipboard.

```
val textBytes = text.toByteArray()
outputStream.write(textBytes)
```

Receiving Clipboard Content

Similar to receiving files, receiving text and images in the clipboard should also work in the background. The reception process is carried out in the `LinkWaveService` (see Section 5.5.3) and is illustrated in Figure 48.

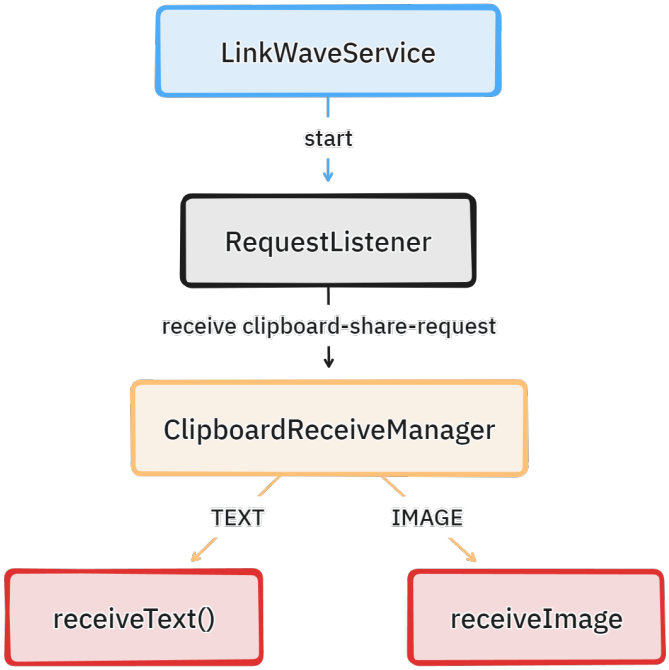


Figure 48: Clipboard Sharing Process on Android

When the RequestListener receives a `CLIPBOARD_SHARE` request, it forwards it to the `ClipboardReceiveManager`. The ClipboardReceiveManager is provided with the `NetworkStream`, through which it receives the type of the content. For text, the content is copied to the clipboard using the `receiveClipboardText` function. For an image, it is copied to the clipboard using the `receiveClipboardImage` function.

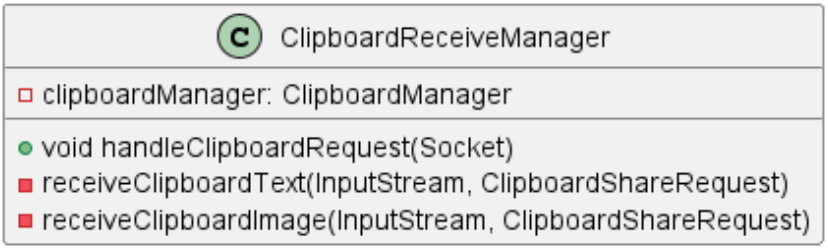


Figure 49: UML of ClipboardReceiveManager

receiveClipboardText() This function reads the text from the `NetworkStream` and copies it to the clipboard. The following code snippet shows the implementation.

```
val textBytes = ByteArray(clipboardShareRequest.length)
val textBytesRead = inputStream.read(textBytes)
text = String(textBytes.sliceArray(0 until textBytesRead))

clipboardManager.setPrimaryClip(
    ClipData.newPlainText("linkwave_text", text)
)
```

receiveClipboardImage() This function reads the image from the `NetworkStream` and copies it to the clipboard. Similar to file transfer, the image is transmitted in chunks. The following code snippet shows the implementation.

```
val uri =
    FileProvider.getUriForFile(
        androidContext,
        "${androidContext.packageName}.fileprovider",
        tempFile
    )
val clipData = ClipData.newUri(androidContext.contentResolver, "
   LinkWave image", uri)
clipboardManager.setPrimaryClip(clipData)
```

The Content Provider is necessary to transfer files to other apps and is defined in the `AndroidManifest.xml` file. The configuration of the `FileProvider` is shown in the following code snippet.

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="${applicationId}.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/file_paths" />
</provider>
```

The `FileProvider` allows apps to access shared images. Which files are allowed to be shared is defined in the `file_paths.xml` file. In this case, only the temporary folder is defined.

5.3.4 On Windows

Clipboard Sharing works in principle like File Sharing but with some differences. Here, the user cannot confirm whether they want to receive the clipboard content. After copying the content, it is automatically sent only to devices logged in with the same account. Clipboard Sharing processes and forwards requests in the same way as File Sharing. Clipboard Sharing is divided into two classes: ClipboardSender and ClipboardReceiver.

ClipboardReceiver

ReceiveTypeAsync This method is the first step in the receiving process and reads the type of shared content (e.g., text or image) from the network stream. It deserializes the received data into a `ClipboardShareRequest` object, which contains information about the type and length of the content. After successfully receiving and deserializing the content information, the method sends an acknowledgment (ACK) back to the sender to signal readiness to receive the actual content data.

```
public async Task ReceiveTypeAsync()
{
    _ = await _networkStream.ReadAsync(receiveTypeBytes);
    // Deserialization of received data
    await LinkWaveSender.SendCommandAsync(RequestTypes.ACK, ...);
}
```

ReceiveClipboardContentAsync This method waits for the transmission of the actual clipboard content after receiving and confirming the content type. It calls `ReceiveClipboardConten` to receive the content bytes. Depending on the type of content, different actions are executed.

```
public async Task ReceiveClipboardContentAsync()
{
    switch (_shareRequest.ClipboardType)
    {
        case ClipboardContentType.Text:
            _ = _uIFunctionStore.SetLastClipboardReceived(text);
            _ = _uIFunctionStore.SetClipboardText(text);
            break;
        case ClipboardContentType.Image:
            // Receiving image data
            _ = _uIFunctionStore.SetLastClipboardReceived(checksum)
                ;
            _ = _uIFunctionStore.SetClipboardImage(bytes);
            break;
    }
}
```

ReceiveClipboardContentBytes After calling `ReceiveClipboardContent`, this method reads the sent content byte by byte and stores it in a byte array. This method ensures that the entire message, regardless of its size, is received efficiently and completely.

```
    private async Task<byte[]> ReceiveClipboardContentBytes(int
       length)
    {
        ...
        while (bytesRead < length)
        bytesRead += await _networkStream.ReadAsync(buffer, ...));
        return buffer;
    }
```

Writing to the Clipboard Reading and writing to the clipboard on Windows is handled in C# by the `System.Windows.Clipboard` class. This class contains static methods for interacting with the clipboard. To set text, the `SetText()` method is used.

```
private async Task SetClipboardTextAsync(string text)
{
    System.Windows.Clipboard.SetText(text);
}
```

When setting images in the clipboard, a slightly more complex approach must be taken. Although there is a `SetImage()` method, it only works with bitmaps. This means that any image formats with transparent backgrounds will lose that transparency. To work around this, a custom `DataObject` must be created. This object adds the standard bitmap field to ensure compatibility with older programs. Additionally, a `PNG` field is created, to which the image in the form of a byte array is added. This `DataObject` is then copied to the clipboard. As a result, images with transparent backgrounds can also be copied to the clipboard, provided the receiving program can read the `PNG` field. Programs like Microsoft Word, Google Docs, or Photoshop can do this.

```
private async Task SetClipboardImageAsync(byte[] imageBytes)
{
    var dataObject = new System.Windows.DataObject();
    var memStream = new MemoryStream(imageBytes)
    var bitmapImage = new BitmapImage();
    bitmapImage.BeginInit();
    bitmapImage.StreamSource = memStream;
    bitmapImage.EndInit();
    dataObject.SetData(DataFormats.Bitmap, bitmapImage, true);
    dataObject.SetData("PNG", memStream, false);
    System.Windows.Clipboard.SetDataObject(dataObject, true);
}
```

ClipboardSender

Communication begins with the sending of an initialization command to start the process. After sending, the sender waits for an acknowledgment (ACK) from the recipient. This acknowledgment ensures that the recipient is ready to receive further data. Once the acknowledgment is received, the clipboard sender transmits detailed information about the clipboard content. After the second acknowledgment, the actual transfer of content begins.

```
public async Task SendAsync(SslStream networkStream)
{
    // Sending initialization command
    while (_contentBytes.Length > 0)
    {
        var chunkSize = Math.Min(...);
        await networkStream.WriteAsync(_contentBytes, 0, chunkSize)
            ;
        _contentBytes = _contentBytes[chunkSize..];
    }
}
```

5.4 User Account

The LinkWave user account is a critical component of the application as it enables user authentication and authorization.

Since LinkWave devices connect automatically, it is important to ensure that unauthorized devices cannot access data. Other alternatives, such as KDE Connect, require users to manually connect devices. To simplify usage, LinkWave eliminates the need for manual connection. LinkWave devices connect automatically when they are on the same network. However, this introduces a security risk, as unauthorized devices could also be on the same network. To prevent this, a user account is required.

Some features, such as the clipboard, are only available to logged-in users because they grant significant control over the devices. Other features are also available without login.

LinkWaveApp user accounts can be created either in the respective apps or on the web dashboard. Users can register with an email address and password or sign in using a Google account.

5.4.1 Functionality

User management is a crucial part of the API as it enables user authentication and authorization.
The implementation includes managing user accounts, authentication via JSON Web Tokens (JWT), and the integration of Google OAuth for authentication services.
In addition, two-factor authentication (2FA) is supported to enhance the security of user accounts.

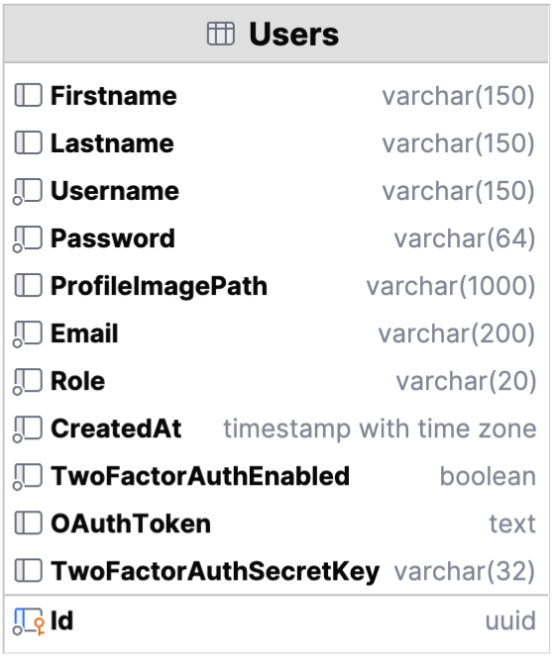


Figure 50: Users Table

The 'Users' table in LinkWave's PostgreSQL database system includes the following fields:

Database Structure

- **Firstname:** The user's first name.
- **Lastname:** The user's last name.
- **Username:** The unique username for login.
- **Password:** The user's encrypted password.

- **ProfileImagePath:** The file path to the user's profile picture.
- **Email:** The user's email address.
- **Role:** The user's role, which defines access permissions.
- **CreatedAt:** The creation date and time of the user account.
- **TwoFactorAuthEnabled:** A boolean indicating whether two-factor authentication is enabled.
- **OAuthToken:** A token for OAuth authentication procedures.
- **TwoFactorAuthSecretKey:** A secret key for two-factor authentication.
- **Id:** The unique ID of the user account.

Each field serves a specific purpose and is critical for authentication, authorization, and managing user information. The database structure supports strong security measures to ensure sensitive data, such as passwords and authentication keys, is securely stored. UUIDs guarantee the uniqueness of each record, while email addresses and profile images contribute to personal identification of users. Role distribution is essential for permission management in the system.

Endpoints

In the LinkWave backend, certain endpoints are central to user management as they form the basis for security and functionality. Among the available endpoints, five are particularly important for basic functions such as user login and registration, as well as the implementation and management of two-factor authentication.

LoginWithEmail

The *LoginWithEmail* function allows users to log in with their email address and password. Upon successful authentication, a JWT is generated and returned, which serves as an authentication token for future requests.

LoginOAuthGoogle

LoginOAuthGoogle handles login via Google OAuth, allowing users to authenticate with their Google account. After successful authentication through Google, a JWT is also issued for further use.

Register

The *Register* function is responsible for registering new users in the system. Users must provide basic information such as email, password, and their name to create a new account.

GenerateQrCodeUri

With *GenerateQrCodeUri*, users can activate two-factor authentication by generating a QR code, which is scanned using an authenticator app. This adds an additional layer of security to their account.

ValidateTwoFactorAuthentication

The *ValidateTwoFactorAuthentication* function validates the code generated by the authenticator app. This step is required to grant access to the system when two-factor authentication is enabled.
These endpoints are explained in more detail in the following sections.

JSON Web Token

A JSON Web Token (JWT) is a compact, self-contained method for securely transmitting information between parties as a JSON object. The information is trustworthy because it is digitally signed. JWTs can be signed using a secret key (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. [14]

Structure of a JWT

- **Header:** The header typically consists of two parts: the type of token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.

- **Payload (Body):** The payload contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims: registered, public, and private claims.
- **Signature:** To create the signature, the encoded header and the encoded payload must be signed using the algorithm specified in the header and secured with a secret key.

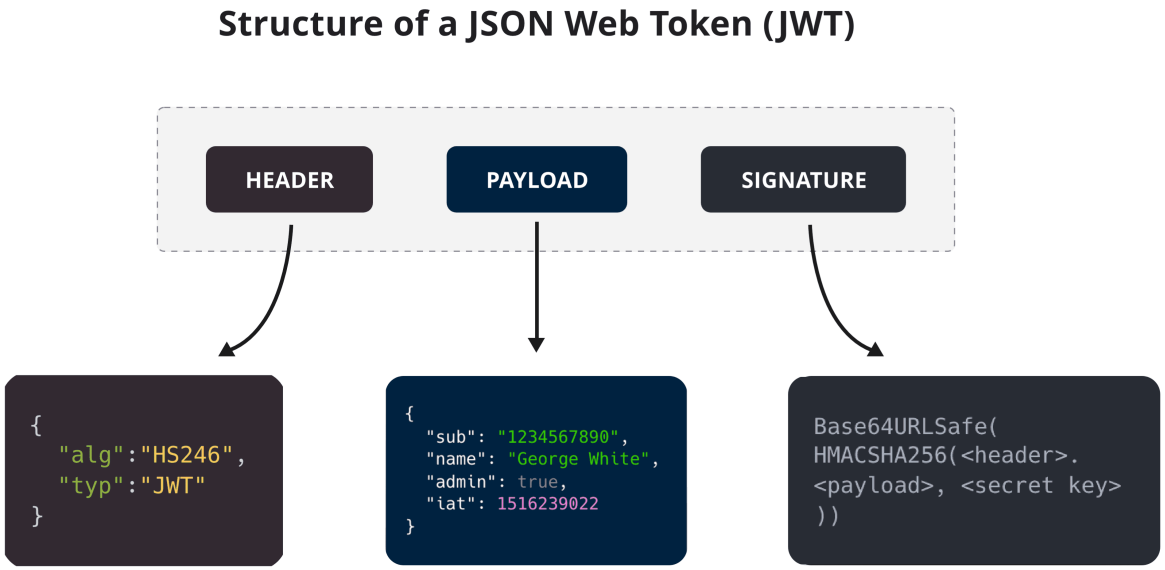


Figure 51: Structure of a JWT

Usage of JWTs During authentication, when a user successfully logs in with their credentials, the authentication server returns a JWT. The client stores this token and sends it with subsequent requests, usually in the Authorization header, to inform the server that the request is authorized.

Refresh Tokens To enhance user experience while maintaining security, refresh tokens are used. An access token typically has a short lifespan. Without a refresh token, users would need to log in again when it expires. A refresh token allows a new access token to be obtained without requiring the user to log in again, enabling a seamless user experience while maintaining security.

Token Generation

- When generating a new token, **claims** for users are created, including their ID, email, role, and token expiration date.
- A **JwtSecurityToken** is generated and contains the issuer, audience, claims, expiration date, and signing credentials.
- The **signing information** is constructed using the security key (SymmetricSecurityKey) and the algorithm used, in this case, HMAC SHA256.
- The final token is then encoded into a string and returned.

Refresh Token Generation

- A refresh token is generated by creating a 32-byte random number and converting it into a Base64 string.
- This token has a longer lifespan than the access token and can be used to obtain a new access token without requiring user authentication.

Security and Validation

By using **claims** and strong encryption, LinkWave ensures both the identity of the user and the integrity of the tokens. The generated tokens comply with security standards and serve as reliable authentication proofs for the duration of their validity.

Google OAuth

LinkWave integrates Google OAuth to provide users with a seamless, secure, and simple login experience. This integration utilizes specific libraries to authenticate with Google's security servers.

What is OAuth?

Google's OAuth flow is a process that allows applications to securely access Google services on behalf of users. The structure of this authentication mechanism is described below and visualized in a diagram.

Figure 52: Google Logo

OAuth is an open standard for access delegation that enables users to securely share private content, such as contact information, over HTTP services. It allows the application to act on behalf of users without requiring their password. OAuth operates with tokens that serve as secure authentication keys, specifying and limiting access rights. It is the preferred method for modern API security and is supported by most major online service providers. [13]

The OAuth Flow

Requesting Permission At the beginning, the application that wishes to access Google services must obtain permission from the users. This is done through a URL leading to a page provided by Google, where users can grant their consent.

The Role of Application Data Important components in this request include:

- **Client ID**: A unique identifier for the application requesting access.
- **Scope**: Defines the scope of access the application is requesting.
- **Redirect URI**: A predefined address to which Google redirects the user after approval.

Consent and Access Token After the user consents via the Google page, the application receives an Authorization Token, which it uses to verify the user's identity and request an Access Token in the next step.

Access Tokens

- **Access Token**: Enables access to Google services for a limited time.
- **Refresh Token**: Can be used to obtain new Access Tokens without requiring the user to log in again.

With the Access Token, the application can act on behalf of the user and access requested services like Gmail, Google Calendar, Classroom, etc.

Renewing Access After the Access Token expires, the application can use the Refresh Token to obtain a new Access Token without disturbing the user.

Below is a diagram illustrating the Google OAuth flow:

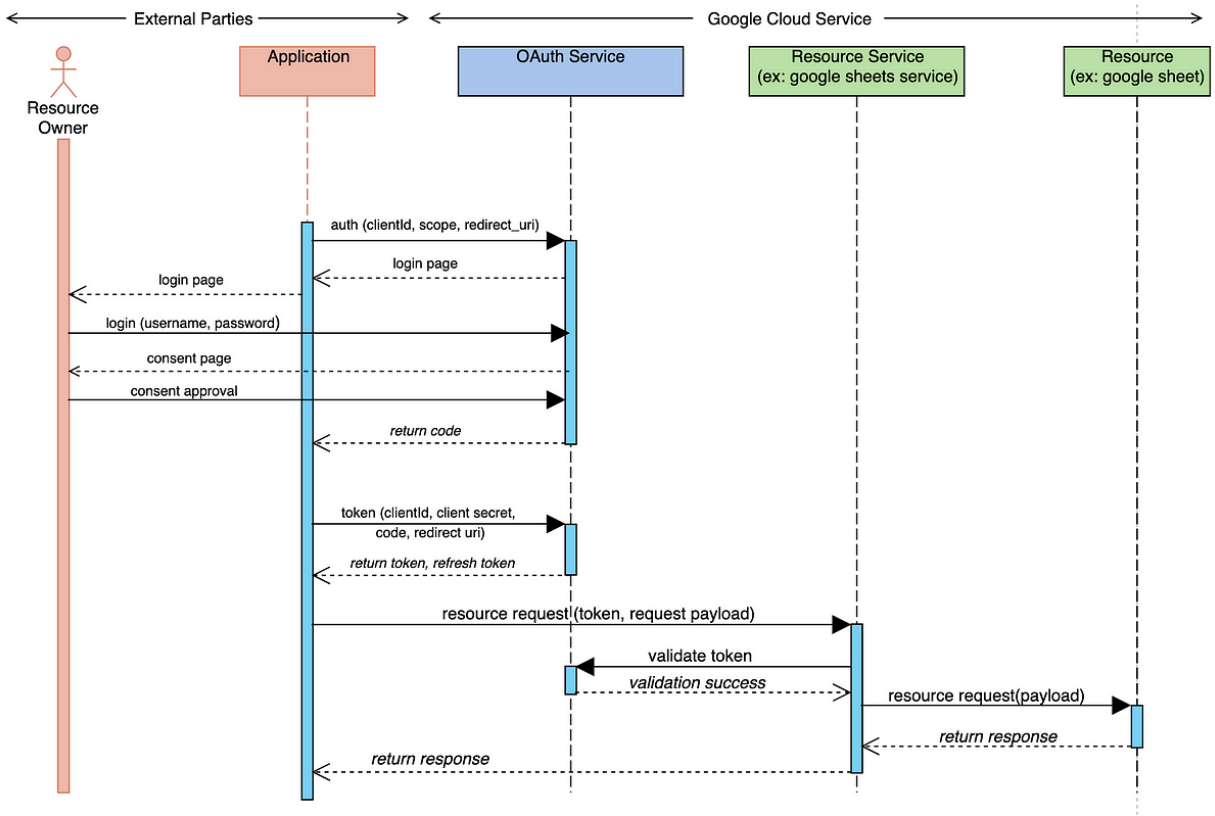


Figure 53: OAUTH Flow

Two-Factor Authentication

Two-Factor Authentication (2FA) is an enhanced security method that goes beyond traditional usernames and passwords by requiring a second step of authentication. This step may involve something the user has, knows, or something biometrically unique. [15]

Two-Factor Authentication Methods

In Two-Factor Authentication (2FA), various methods are used to verify the user's identity. Typically, a One-Time Password (OTP) is used, which can be either time-based (TOTP) or counter-based (HOTP). These methods utilize cryptographic functions like HMAC to ensure that each password is unique and cannot be reproduced. [16]

Time-based One-Time Password (TOTP)

- TOTP is an algorithm that generates a temporary password that changes at regular, short time intervals.
- The authentication servers and the user's authenticator app are synchronized and use the current time as the basis for the password.
- TOTPs provide strong protection against replay attacks because even if a password is intercepted, it quickly becomes invalid.

HMAC-based One-Time Password (HOTP)

- HOTP generates authentication codes based on a secret key and a counter.
- Both the server and the user's authentication device maintain the same counter, which is incremented after each use.
- HOTP tokens are not time-dependent, making them useful in situations without a reliable time source.

Keyed-Hash Message Authentication Code (HMAC)

- HMAC is a specific type of Message Authentication Code (MAC) that combines a cryptographic hash function with a secret cryptographic key.
- In 2FA systems, HMAC is used to generate the one-time passwords required for TOTP and HOTP.
- The use of HMAC ensures the integrity and authenticity of messages (in this case, the passwords).

Two-Factor Authentication in LinkWave

In LinkWave, two-factor authentication (2FA) has been implemented in the backend to enhance the security of user accounts. This section describes the technical steps of the implementation and how users can utilize 2FA.

Two-Factor Authentication Flow

To enable 2FA, users generate a QR code through the web, desktop, or mobile app:

1. Users initiate the 2FA setup in their account settings.
2. A unique secret key is generated on the server side.
3. Using the key, a QR code is created, containing the identification data for the user's 2FA app.
4. Users scan this QR code with an authenticator app installed on their mobile device.
5. The authenticator app stores the information and begins generating time-based one-time passwords (TOTP).

Validating Two-Factor Authentication

During each login attempt, users must provide a TOTP code in addition to their password:

1. Users enter their password and the TOTP code generated by the authenticator app into the login interface.
2. The LinkWave server receives the TOTP code and verifies it using the stored secret key.
3. If validation is successful, users are granted access to their account.

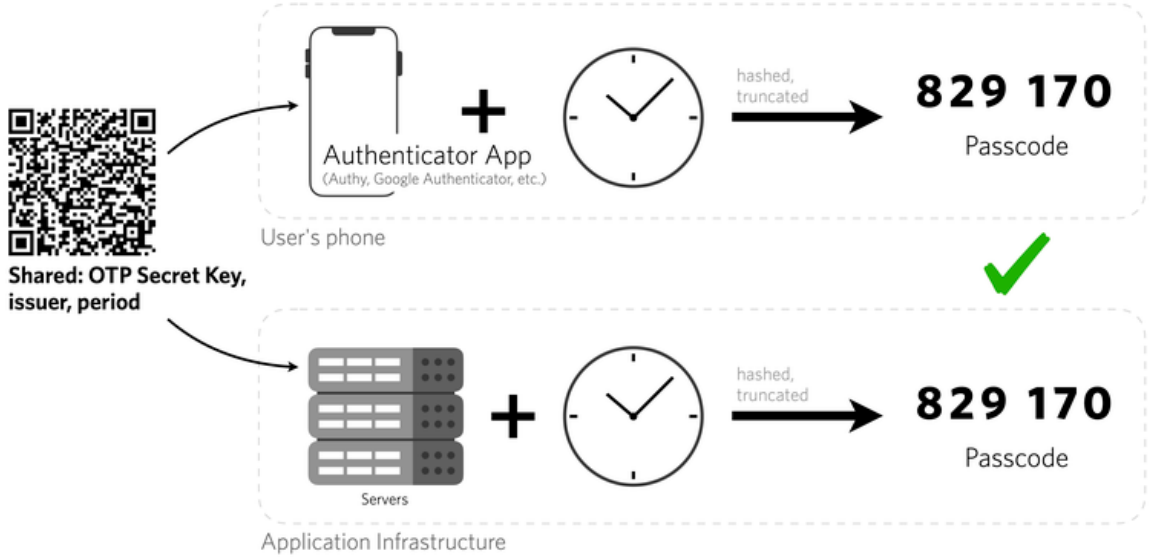


Figure 54: 2FA Flow

5.4.2 On iOS and macOS

The user account is managed in the app on iOS and macOS. Users can register, log in, and edit their profiles. It is possible to log in using an email address and password or a Google account. These options are implemented very differently.

Email and Password

Logging in and registering with an email and password is done via a GraphQL connection to the LinkWave server. The `swift-graphql` library is used to work with GraphQL. Documentation for the functions used can be found on the website [2]. This library enables the creation and sending of GraphQL queries and mutations. Additionally, it supports generating code from GraphQL schemas, which simplifies the use of GraphQL in Swift. The schema is generated with the following command:

```
$ swift-graphql https://api.linkwave.org/graphql/ \
      -o generated.swift
```

Functions and classes are then generated, which can be used with the `swift-graphql` library. To log users in, a `loginWithEmail` request is sent.

```
let query = Objects.Mutation.loginWithEmail(
  email: email,
  password: password)
let request = URLRequest(url: linkWaveHelper.graphqlClient!)
  .querying(query)
let task = URLSession.shared.dataTask(with: request)
{ data, _, _ in
  guard let result = try? data?.decode(query) else { return }
  // Save token
}
```

The JWT is now stored in the `token` variable, which is required for authenticating further requests. The token is also stored in the Keychain to retain it after restarting the app while keeping it secure. This is achieved using the `KeychainSwift` library [3].

```
let token = result.data!
let keychain = KeychainSwift()
keychain.set(token, forKey: "JWT")
```

Registering works conceptually the same way as logging in, except that instead of the `loginWithEmail` mutation, a `register` mutation is sent. This mutation also requires the username, first name, and last name of the user.

Google Account

The login using a Google account (OAuth) on iOS and macOS is implemented through the `GoogleSignIn` library. This library provides a simple `signIn` function, which guides the user through the login process in an open browser window. After the user has logged in, a callback function is called, which retrieves the JWT from the server using the `accessToken`.

```
GIDSignIn.sharedInstance.signIn(withPresenting: mainWindow)
{ signInResult, _ in
  guard let result = signInResult else { return }
  var accessToken = result.user.accessToken
  var query = Objects.Mutation.loginOAuthGoogleAccessToken(
    accessToken: accessToken.tokenString)
}
```

The JWT is then saved in the Keychain again to be used for further requests.

5.4.3 On Android

User management on Android is handled by the Android app. Here, users can register and log in with an email and password. The GraphQL API of the LinkWave server is used for this. The `Apollo-Android` library is used for communication with the GraphQL API. This library allows the creation and sending of GraphQL queries and mutations. It also supports code generation from GraphQL schemas.

Email and Password

To log in users, a `loginWithEmail` request is sent to the server. This request contains the user's email and password. The server's response includes the JWT token, which is used for authentication in future requests. The GraphQL request is illustrated in the following code example.

```
mutation LoginWithEmail($email: String!, $password: String!) {
    loginWithEmail(email: $email, password: $password)
}
```

Apollo-Android automatically generates a class from this mutation, which contains the request. This class is used to send the request to the server. The server responds with the JWT token, which is stored in SharedPreferences. The following code demonstrates how the request is sent to the server and the response is processed.

```
val response = apolloClient.mutation(LoginWithEmailMutation(email,
   password)).execute()

val token = response.data?.loginWithEmail

securePreferences.saveSessionToken(token)
```

The `SecurePreferences` class saves the JWT token in the EncryptedSharedPreferences. SharedPreferences allow data to be stored in a key-value format. EncryptedSharedPreferences encrypt the data before storing it. For future requests, the JWT token is loaded from EncryptedSharedPreferences and added as an Authorization header to requests to the server via an `HTTP-Interceptor`.

```
fun provideApolloClient(
    securePreferences: SecurePreferences
): ApolloClient {
    val okHttpClient = OkHttpClient.Builder()
        .addInterceptor { chain ->
            val original = chain.request()
            val builder = original.newBuilder().method(original.
               method, original.body)
            builder.header("Authorization", "Bearer␣${
               securePreferences.getSessionToken()}")
            chain.proceed(builder.build())
        }
        .build()

    return ApolloClient.Builder()
        .serverUrl("https://api.linkwave.org/graphql/")
        .okHttpClient(okHttpClient)
        .build()
}
```

5.4.4 On Windows

User management on Windows is handled by the Windows app. Here, users can register, log in, and edit their profiles. It is possible to log in using an email address and password or a Google account. For this purpose, the GraphQL API of the LinkWave server is used. Requests are sent via the `GraphQLHttpClient` class from `GraphQL.Client`. GraphQL queries and mutations are used to perform user actions. To log in users, a `loginWithEmail` request is sent, or a `LoginOAuthGoogleAccessToken` request is used if users sign in with Google. Upon successful login, the JWT token is stored in Local Settings.

Registration Process

During the registration process, users enter their personal information, including username, first name, last name, email address, and password. This data is sent to the server via a GraphQL request. Upon successful registration, the system automatically initiates the process of generating an SSL certificate for the user's device. This certificate is used to authenticate the device's identity within the network and establish an encrypted connection to other devices.

The `Register` mutation is used for registration, sending user data to the server and starting the registration process. After successful registration, a JWT token is returned, which is used to authenticate users in future requests. The JWT token is stored in "Local Settings."

```
public async Task RegisterAsync(User user)
{
    string username = UsernameTextBox.Text;
    string email = EmailTextBox.Text;
    // Additional user data
    string password = PasswordBox.Password;
    ...
    var signUpRequest = new GraphQLRequest
    {
        Query = // GraphQL Query to register user
        Variables = // Variables for the query
    };
    var signUpResponse = await graphQLClient.SendQueryAsync<
       SignUpResponseData>(signUpRequest);
    deviceCertificate = await Task.Run(async () => await
       Certificate.CompleteCertificateSigningProcess());
    // Further steps
    App.GetService<ILocalSettingsService>().SaveSettingAsync("
       JWTToken", loginResponse.Data.Token)
}
```

Login Process

Users can log in using their email address and password or their Google account via Google OAuth. During the login process, users enter their credentials, which are sent to the server to verify their identity. If the credentials are correct, a JWT token is returned and used to authenticate users for future requests. After a successful login, the JWT token is stored in the “Local Settings.”

If the user already has an account but logs in on a device for the first time, their device is automatically registered, and an SSL certificate is generated for the device.

The `LoginWithEmail` mutation is used for user login.

```
private async void LoginButton_Click(object sender, Microsoft.UI.
   Xaml.RoutedEventArgs e)
```

```
{
    var loginRequest = new GraphQLRequest
    {
        Query = // GraphQL query to log in the user,
        Variables = // Variables for the query
    };
    var loginResponse = await graphQLClient.SendQueryAsync<
       LoginResponse>(loginRequest);
}
```

Google OAuth is used for user login. Due to its complexity and the many steps involved, not all methods are detailed here.

```
string authorizationRequest = // Authorization request
// Launch browser with the authorization request
private async Task<string?> performCodeExchangeAsync(string code,
   string code_verifier)
{
    var loginRequest = new GraphQLRequest
    {
        Query = // GraphQL query to log in the user,
        Variables = // Variables for the query
    };
    GraphQLResponse<LoginResponse> loginResponse;
    loginResponse = await graphQLClient.SendQueryAsync<
       LoginResponse>(loginRequest);
    return loginResponse.Data.Token;
}
// Additional steps
```

5.4.5 On the Web

The web dashboard allows users to manage their accounts. Here, they can register, log in, and edit their profiles. It is possible to log in using an email address and password or a Google account. For this, the GraphQL API of the LinkWave server is used.

The library `graphql-request` [18] is used to communicate with the GraphQL API. The dashboard is written in SvelteKit. Since SvelteKit is a server-side rendering framework, the GraphQL API is directly called from the server. The SvelteKit server acts as a proxy for the GraphQL API.

Email and Password

To log in users, a `loginWithEmail` request is sent to the server. This request contains the user's email and password. The email and password are sent via form actions to the SvelteKit server. The server sends the request to the LinkWave server and receives the JWT token in return. This token is stored in a cookie for future requests.

```
const loginRes = await gqlClient.request(loginWithEmailMutation, {
    email: form.data.email,
    password: form.data.password
});
const sessionToken = loginRes.loginWithEmail;

event.cookies.set('sessionToken', sessionToken, {
    httpOnly: true,
    maxAge: 60 * 60 * 24 * 7,
    path: '/',
    sameSite: 'lax',
    secure: !dev
});
```

- **httpOnly**: The cookie cannot be accessed via client-side JavaScript. This enhances security as the JWT token cannot be stolen through XSS attacks.

- **maxAge**: The cookie expires after one week. Users will need to log in again after this period.
- **path**: The cookie is available across the entire website.
- **secure**: The cookie is only sent over HTTPS when the website is in production mode.

Google Account

Logging in with a Google account (OAuth) on the web dashboard works via the `google-auth-libra` Users are redirected to the Google OAuth page, where they can log in with their Google account. After logging in, they are redirected to a callback URL containing the authorization code as a query parameter. This authorization code is sent to the LinkWave GraphQL API.

```
google: async () => {
    const redirectUri = googleOAuthClient.generateAuthUrl({
        accessType: 'offline',
        scope: [
            'https://www.googleapis.com/auth/userinfo.email',
            'https://www.googleapis.com/auth/userinfo.profile'
        ]
    });

    return redirect(302, redirectUri);
}
```

The callback URL is defined on the SvelteKit server. SvelteKit provides the ability to define so-called server routes, which are simple REST endpoints. After users log in with their Google account, they are redirected to such a server route. This route reads the authorization code from the query parameters and sends it to the LinkWave GraphQL API.

```
export const GET: RequestHandler = async (event) => {
    const code = event.url.searchParams.get('code');
```

```
    const loginRes = await gqlClient.request(
       loginWithGoogleDocument, {
        authorizationCode: code
    });
    const sessionToken = loginRes.loginOAuthGoogle;
    setSessionCookie(event, sessionToken);
    return redirect(302, '/');
};
```

5.4.6 On the Server

JWT Implementation

LinkWave implements authentication using JSON Web Tokens (JWT) on the .NET platform. The implementation uses core classes from the `System.IdentityModel.Tokens.Jwt` package and supporting security classes.

JwtProvider Class The `JwtProvider` class is responsible for generating JWTs and refresh tokens. It is designed to obtain settings from the `JwtSettings` configuration and apply them to generate tokens.

```
private JwtSettings _settings;
public JwtProvider(IOptions<JwtSettings> settings)
{
    _settings = settings.Value;
}
public string GenerateToken(User user)
{
    var claims = new[]
    {
        new Claim("UserId", user.Id.ToString()),
        new Claim(ClaimTypes.Email, user.Email),
        ...
    };
    var token = new JwtSecurityToken(
        issuer: _settings.Issuer,
        audience: _settings.Audience,
        ...
    );
    var tokenValue = new JwtSecurityTokenHandler().WriteToken(token
       );
    return tokenValue;
}
```

OAuth Implementation

LinkWave uses the Google OAuth API for authentication services to ensure secure verification of user identities and the safe retrieval of their information.

Token Verification and User Data Retrieval

The `GoogleOAuth` class encapsulates the logic for interacting with Google's OAuth 2.0 endpoint. The main process involves:

1. Creating a `RestClient` targeting `"https://oauth2.googleapis.com"`.
2. Configuring a `RestRequest` to request the token via the POST method.
3. Using application data such as `redirectUri`, `clientId`, and `clientSecret`, dynamically loaded from settings based on the `clientType`.
4. Returning a response from the Google server as a string.

Below is a code snippet demonstrating the verification of a Google token:

```
public async Task<string?> VerifyGoogleTokenAsync(...)
{
    ...
    RestClient client =
    new RestClient("https://oauth2.googleapis.com");
    RestRequest request = new RestRequest("token", Method.Post);
    ...
    return response.Content;
}
```

Adjustments for Mobile OAuth Authentication

The `LoginOAuthGoogleAccessToken` method was specifically implemented to support the OAuth process on mobile devices, as these devices only work with access tokens and do not return an authorization code.

Summary of Login Processes

Two primary methods, `LoginOAuthGoogle` and `LoginOAuthGoogleAccessToken`, allow users to log in via Google OAuth. The standard process uses the authorization code, while the alternative method directly works with the access token.

```
public async Task<string?> LoginOAuthGoogle(...)
{
    ...
    GoogleTokenResponse accessToken = ...
    Database.Models.User? userData =
    _googleOAuth.GetUserData(accessToken.AccessToken);
    ...
}

public async Task<string?> LoginOAuthGoogleAccessToken(...)
{
    ...
    Database.Models.User userData =
    _googleOAuth!.GetUserData(accessToken);
    ...
}
```

In both cases, the user data is extracted from the Google account information and stored in the database. Subsequently, a JWT is generated for the users and returned, allowing them to authenticate themselves.

Two-Factor Authentication Implementation

The technical implementation on the server is carried out through two main functions:

```
public async Task<string> GenerateQrCodeUri(...)
{
    var twoFactorAuthenticator = new TwoFactorAuthenticator();
    var secretKey = Utilities.GenerateRandomString(32).Result;
    ...
```

```
    // Generate QR code
    ...
    return setupInfo.QrCodeSetupImageUrl;
}

public async Task<bool> ValidateTwoFactorAuthentication(...)
{
    ...
    // Validation of the TOTP code
    ...
    return twoFactorAuthenticator.ValidateTwoFactorPIN(secretKey,
       token);
}
```

5.5 Background Execution

Background execution is a very important but invisible feature for the user. It allows the app to continue running in the background, receiving and sending data. LinkWave should feel like an integral part of the operating system for users.

Requests for file transfers should also be processed without the app being open. If a user wants to send a file, they should be able to do so even if the app is closed. Additionally, the clipboard should be shared without any interaction with the app. Ideally, a user should not even notice that the app is running in the background. The app is essentially only needed for configuration purposes.

However, a user should be aware that the app is running in the background. It should also be easy to completely close the app or open the settings. A user should be able to use the app as an extension of the operating system, creating an experience similar to that of an ecosystem.

5.5.1 On iOS

The implementation of background execution on iOS is not included in the current version of LinkWave. This is because apps on iOS devices operate within a sandbox. Apple does not allow apps to run continuously in the background outside of this sandbox.

Apple requires apps to use the background update paradigm. This means that apps define an update function, which is called by iOS at regular intervals. This interval can be dynamically adjusted by iOS. Within this function, apps are expected to update data and send notifications before being put back to sleep. (see Apple Developer Documentation [5])

Since LinkWave is designed to continuously transfer data, putting it to sleep would severely limit its functionality. Therefore, implementing background execution on iOS is not feasible. However, users can still utilize all app features while the app is open.

5.5.2 On macOS

Unlike iOS, macOS allows for the implementation of background processes, even within a sandbox. This is achieved by ensuring that a UI element is continuously present. A menu bar icon can be used as the UI element (see Chapter 3.2). This icon also allows users to close the app or open the settings via a context menu. A menu bar icon can be easily created using the `MenuBarExtra` element in a SwiftUI view.

```
MenuBarExtra {
  MenuBarExtraContent() // Content of the context menu
} label: {
  Image("menubar-icon") // Menu bar icon image
}
```

5.5.3 On Android

On Android, long-running background processes are implemented using `Services`. There are two types of services: foreground services and background services. Since Android 8.0 (API Level 26), restrictions on background services have been introduced to improve battery life. These restrictions have essentially made background services obsolete. Instead, foreground services should be used, which display a notification to indicate that a service is running in the background. The `LinkWaveService` is a foreground service and includes functions to be executed in the background, such as receiving files, receiving shared clipboard content, and advertising the device on the network via Bonjour. The following figure shows the class diagram of the `LinkWaveService`.

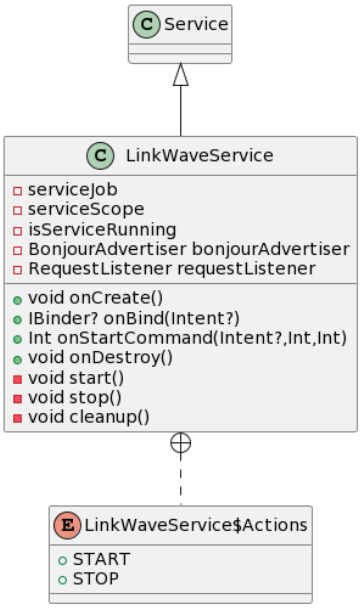


Figure 55: LinkWaveService Class Diagram

The `onStartCommand()` function is called when the service receives an Intent. This Intent can either be a `START` or `STOP` action. As the names suggest, the service is started with a `START` action and stopped with a `STOP` action.

start() In the `start()` function, the `BonjourAdvertiser` is started, advertising the device on the network, and the `RequestListener`, which waits for requests from other devices, is launched. Additionally, a notification is created to signal to the user that the

service is running in the background.

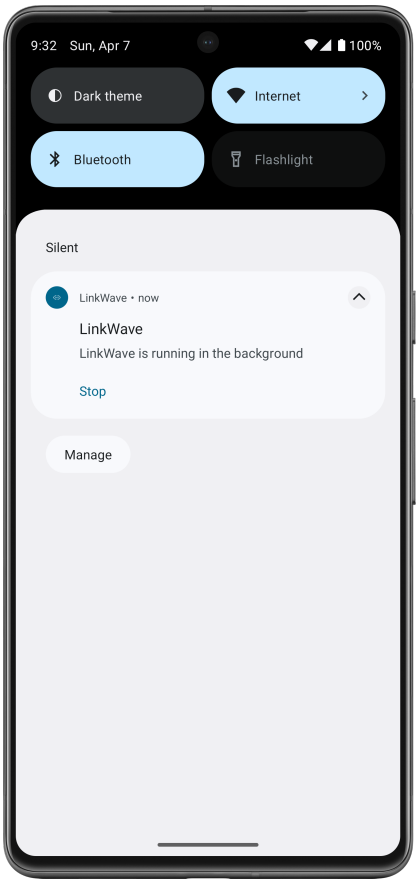


Figure 56: LinkWaveService Notification

stop() In the `stop()` function, the `BonjourAdvertiser` and the `RequestListener` are stopped. All coroutines running in the service are canceled, and the notification is removed.

5.5.4 On Windows

Similar to macOS, background processes can be implemented on Windows using a tray icon. This icon is displayed in the taskbar and allows users to close the app or open the settings. The tray icon is created using the `NotifyIcon` class. This class enables an icon to be displayed in the taskbar and respond to user interactions.

```
var trayIcon = new NotifyIcon();
trayIcon.Text = "LinkWave";
trayIcon.Icon = new System.Drawing.Icon(Path.Combine(
    AppContext.BaseDirectory, "Assets/TrayIcon.ico"));
trayIcon.Visible = true;
```

To create a context menu, the `ContextMenuStrip` class is used. This menu is assigned to the `NotifyIcon` and is displayed when users right-click on the icon.

```
var trayMenu = new ContextMenuStrip();
trayMenu.Items.Add("Open", null, OnOpenWindow);
trayMenu.Items.Add("Open␣Settings", null, OnOpenSettings);
trayMenu.Items.Add("Send␣File", null, OnOpenSendFile);
trayMenu.Items.Add("Quit", null, OnQuit);
trayIcon.ContextMenuStrip = trayMenu;
```

By overriding the `Closed` method, it is ensured that closing the main window does not automatically terminate the app. In this method, the main window is only hidden instead of closed. Additionally, the operating system is informed that the operation was successful, preventing the app from being terminated by the OS.

```
private void MainWindow_Closed(object sender, WindowEventArgs e)
{
    e.Handled = true;
    this.Hide();
}
```

5.6 Encryption

Encryption is a very important part of LinkWave. Since LinkWave is intended to be used in insecure networks, such as free Wi-Fi in cafés, it is crucial that the data is transmitted securely. It must not be possible for an attacker to intercept files sent via LinkWave. Additionally, it should not be apparent when files are being transferred or what files are being sent.

At the same time, the data should be transferred very quickly, without noticeable delay. Therefore, the encryption must be very efficient. Additionally, the encryption should be based on a widely used standard to ensure secure and identical implementation across all platforms.

LinkWave uses TCP for all connections. However, TCP does not handle data encryption. Thus, LinkWave itself must ensure encryption.

5.6.1 Functionality

LinkWave has its own Certificate Authority (CA), which is used to sign certificates. These certificates are utilized to encrypt communication between devices. This process is illustrated in the following diagram. It is important to note that steps one and two are only performed during the initial login or registration on a device. The certificates are then stored securely on the device, such as in the Certificate Store on Windows or the Keychain on macOS. These certificates are later used to encrypt communication between devices.

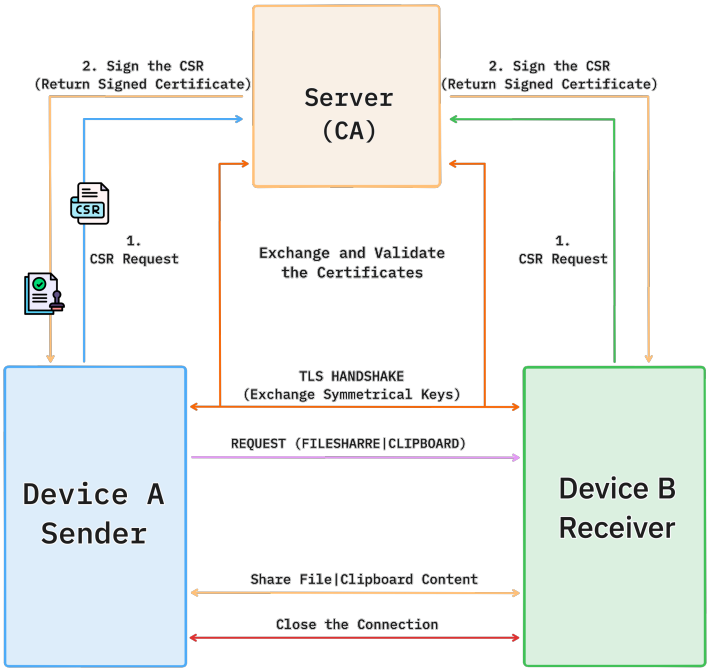


Figure 57: SSL/TLS

1. **CSR Creation:** Devices A and B generate a Certificate Signing Request (CSR) and send it to the server.

2. **CSR Signing:** The server signs the CSR and issues an SSL certificate, which is sent back to Devices A and B.

3. **TLS Handshake:** Using the received certificates, Devices A and B perform a TLS handshake. Certificates are exchanged and validated.

4. **Symmetric Key Exchange:** During the handshake, a symmetric encryption key is generated.

5. **Start of Encrypted Communication:** All subsequent data is transmitted encrypted with the symmetric key.

6. **Connection Termination:** After data transfer, the connection is securely terminated while maintaining the security of the transferred data.

What is SSL/TLS?

About SSL and TLS SSL and TLS are cryptographic protocols developed to enable secure data transmission over insecure networks such as the Internet. Originally developed by Netscape, SSL quickly became the standard for secure web communication before being replaced by TLS, its improved and more secure successor. Both protocols provide mechanisms for encrypting and authenticating data exchanged between web browsers and web servers and are also used in other protocols like email, VoIP, and instant messaging.

The SSL/TLS Handshake Establishing an SSL/TLS-secured connection is a multi-step process known as the “Handshake.” This process lays the foundation for secure communication by verifying the identity of the communication partners and establishing a shared encryption key.

Agreement In the first step of the handshake, the client and server agree on the protocol version to be used and select algorithms for encryption and authentication. This selection ensures that both parties use the same cryptographic standards, which are essential for secure data transmission.

Key Exchange After agreeing on the protocols and algorithms, the communication partners exchange key information. This step often involves authenticating the server (and sometimes the client) using certificates. Certificates issued by a trusted Certification Authority (CA) enable the parties to verify each other’s identity and generate a shared symmetric encryption key.

Encrypted Communication With the successful completion of authentication and key exchange, encrypted communication begins over the secured channel. All transmitted data is encrypted using the agreed-upon symmetric key, ensuring confidential and integral data transmission.

The Importance of Certificates Certificates play a central role in the SSL/TLS handshake. They contain the certificate holder's public key and identity information and are signed by a Certification Authority. These certificates are used to verify the server's identity and prevent man-in-the-middle attacks, where attackers attempt to intercept communication.

Source: [8]

Implementation

The encryption process is explained in detail below:

5.6.2 On iOS and macOS

Encryption on iOS and macOS is achieved using SSL/TLS certificates. These can be easily passed to an `NWParams` object and attached to an `NWConnection`. This object is then used to establish a connection to a server, which is automatically encrypted.

The first step involves generating a public-private key pair. This can be done using the `SecKeyCreateRandomKey` function, which also stores the key pair in the Keychain.

```
let tag = tlsKeysTag.data(using: .utf8)!
let attributes: [String: Any] = [
    kSecAttrKeyType as String: kSecAttrKeyTypeRSA,
    kSecAttrKeySizeInBits as String: 2048,
    kSecPrivateKeyAttrs as String: [
        kSecAttrIsPermanent as String: true,
        kSecAttrApplicationTag as String: tag,
    ],
]
var error: Unmanaged<CFError>?
guard let privateKey = SecKeyCreateRandomKey(attributes as
   CFDictionary, &error) else { return }
let publicKey = SecKeyCopyPublicKey(privateKey)
```

In the second step, a certificate must be generated and signed by the server. This certificate is then used for encryption. A signing request is known as a CSR (Certificate Signing Request). This CSR is created using the `CertificateSigningRequest` library. The CSR is sent to the server, which signs it and sends it back. The `signCertificate` mutation is used for this purpose.

```
let csr = CertificateSigningRequest(
  commonName: commonName,
  organizationName: organizationName,
  organizationUnitName: organizationName, countryName: countryName,
  keyAlgorithm: algorithm
)
let publicKeyBits =
  SecKeyCopyExternalRepresentation(publicKey, nil)! as Data
let builtCSR = csr.buildCSRAndReturnString(
  publicKeyBits, privateKey: privateKey, publicKey: publicKey)
var query = Objects.Mutation.signCertificate(csrPem: builtCSR)
```

This certificate is stored in the Keychain to be used later for encryption. This can be achieved with the `SecItemAdd` function. The certificate must also be labeled for easy retrieval.

```
guard let derData = Data(base64Encoded: pemBody) else {
    return false }
guard let certificate = SecCertificateCreateWithData(nil, derData
   as CFData) else {
    return false }
let addquery: [String: Any] = [
    kSecClass as String: kSecClassCertificate,
    kSecValueRef as String: certificate]
var status = SecItemAdd(addquery as CFDictionary, nil)
if status == errSecSuccess { } else { return }
status = SecCertificateSetPreferred(
    certificate, TLSCertLabel as CFString, nil)
```

To establish an encrypted connection, a `TLSOptions` object must first be created. This object contains the certificates used for encryption. It also specifies the validation of certificates from other parties.

```
let identity = retrieveIdentityFromKeychain()
let tlsOptions = NWProtocolTLS.Options()
guard let os_sec_identy = sec_identity_create(identity) else
{ return }
sec_protocol_options_set_local_identity(
    tlsOptions.securityProtocolOptions, os_sec_identy)
sec_protocol_options_set_min_tls_protocol_version(
    tlsOptions.securityProtocolOptions, .TLSv12)
sec_protocol_options_set_verify_block(
    tlsOptions.securityProtocolOptions,
{ sec_protocol_metadata, sec_trust, sec_protocol_verify_complete in
  // Validate certificate
}, DispatchQueue.main)
```

In the function for certificate verification, the certificate is sent to the server. The server then responds with whether the certificate is valid. If the certificate is valid, the connection is established. Otherwise, the connection is terminated. This is achieved using the `isCertificateValid` query.

```
let secTrust = sec_trust_copy_ref(sec_trust).takeRetainedValue()
let serverCertificates = SecTrustCopyCertificateChain(secTrust)
let serverCertificate = serverCertificates[0] as SecCertificate
let pemCert = convertToPEMCertificate(serverCertificate)
let query = Objects.Query.isCertificateValid(certPem: pemCert)
```

Using these `TLSOptions`, an `NWConnection` can now be created. This connection is encrypted and can be used for data transmission.

```
let tlsOptions = getTLSOptions()
let params = NWParameters(tls: tlsOptions, tcp: .init())
let connection = NWConnection(to: endpoint, using: params)
```

5.6.3 On Android

Encryption on Android is achieved by using SSL/TLS certificates. These certificates are used to encrypt communication between devices. The process begins with generating a key pair and creating a CSR (Certificate Signing Request). This CSR is sent to the server using the `SignCertificate` mutation, where it is signed, and an SSL certificate is issued. This certificate is then stored on the device and used to encrypt communication.

```
mutation SignCertificate($csrPem: String!) {
    signCertificate(csrPem: $csrPem)
}
```

Key Generation

The `SSLHelper` object provides methods for key generation and certificate requests. The `generateAndStoreKeyPair()` method generates a key pair and stores it in the Android KeyStore. The `generatePemCsr` method creates a certificate request and returns it as a PEM string.

```
val keyPair = generateAndStoreKeyPair("linkwave_keypair")
val subject =
    X500Name("CN=${LocalDeviceInfo.name}, O=linkwave.org")
val csrBuilder = JcaPKCS10CertificationRequestBuilder(
    subject,
    keyPair.public
)
val signer = JcaContentSignerBuilder("SHA256withRSA").build(keyPair
   .private)
val csr = csrBuilder.build(signer)

return csrToPem(csr)
```

Certificate Signing

To sign the certificate request, the `SignCertificate` mutation is sent to the server. The server signs the request and returns the signed certificate as a PEM string.

```
val response = apolloClient.mutation(SignCertificateMutation(csr))
    .execute()
val signedCertificate = response.data?.signCertificate ?: throw
   Exception("No signed certificate found")
SSLHelper.saveSignedCertificateToKeystore(signedCertificate)
```

The signed certificate is saved in the Android KeyStore and can be used for encryption of communication.

Encrypted Communication

Encrypted communication is achieved through the use of the `SSLContext` object. This object is initialized with the signed certificate and can then be used to create `SSLSocket` objects. The following code shows the creation of an `SSLContext` object with the signed certificate.

```
val privateKeyEntry = SSLHelper.getPrivateKeyEntry()
val keyManagerFactory = KeyManagerFactory.getInstance(
   KeyManagerFactory.getDefaultAlgorithm()).apply {
    init(KeyStore.getInstance(KeyStore.getDefaultType()).apply {
        load(null)
        setKeyEntry("alias", privateKeyEntry.privateKey, null,
           privateKeyEntry.certificateChain)
    }, null)
}

val sslContext = SSLContext.getInstance("TLS").apply {
    init(keyManagerFactory.keyManagers, null, null)
}
```

Now the `SSLContext` object can be used to create an SSLFactory, which can then be used to create `SSLSocket` objects. val sslSocketFactory = sslContext.socketFactory val

sslSocket = sslSocketFactory.createSocket() as SSLSocket

This `SSLSocket` can then be used for secure communication with other devices.

```
val outputStream = sslSocket.outputStream
val inputStream = sslSocket.inputStream

outputStream.write("Hello, World!".toByteArray())
val response = inputStream.read()
```

5.6.4 On Windows

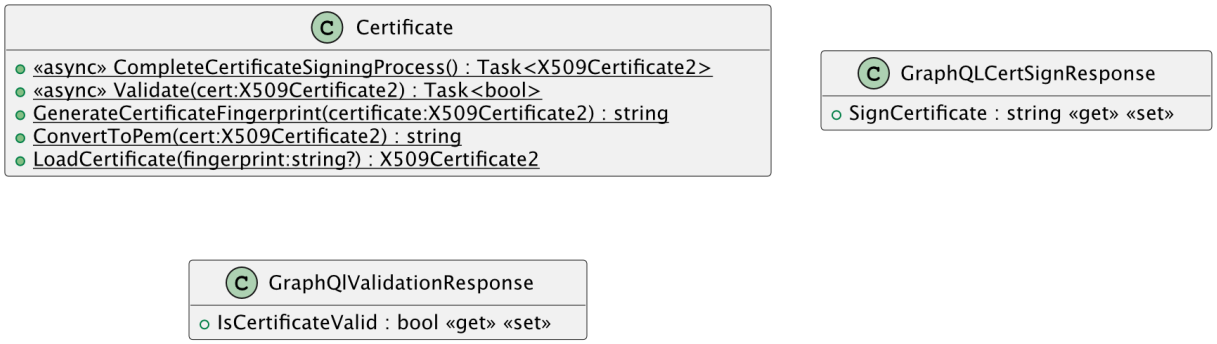


Figure 58: UML Certificate

Certificate Class

The `Certificate` class in LinkWave includes methods for generating certificate signing requests (CSR), interacting with the GraphQL API to sign the requests, and validating certificates.

The `CompleteCertificateSigningProcess` method coordinates the creation and signing of a certificate, while the `Validate` method checks the validity of the received certificate.

```
public static async Task<X509Certificate2>
   CompleteCertificateSigningProcess()
{
    var cert = GenerateCertificateSigningRequest();
    var signedCertPem = await SendCsrForSigning(cert.csr);
    var signedCertBytes = PemToDer(signedCertPem);
    var importedCert =
    ImportSignedCertificate(signedCertBytes, cert.privateKey,
       GenRandomPassword());
    return importedCert;
}

public static async Task<bool> Validate(X509Certificate2 cert)
```

```
{
    var certPem = ConvertToPem(cert);
    return await ValidateCertificateByServerAsync(certPem);
}
```

1. **Certificate Signing Request (CSR)**: This process describes the request for a certificate by a device. A certificate request is sent to the Certification Authority (CA). After verification, the CA signs the request and issues a certificate, which the device uses to authenticate its identity. [11]

2. **X509Certificate2**: This class is part of the .NET library and is used to manage X.509 certificates. Its functions include importing, exporting, and validating certificates. [12]

SSL Class

The SSL class is responsible for establishing and ensuring SSL/TLS connections. It includes methods for both the server and the client to perform SSL handshakes and establish a secure connection.

```
public static async Task<SslStream>
   AcceptAndAuthenticateClientAsync(...)
{
    TcpClient client = await listener.AcceptTcpClientAsync();
    SslStream sslStream = new SslStream( client.GetStream(),
    leaveInnerStreamOpen: false,
    new RemoteCertificateValidationCallback(ValidateCertificate)
       ,...);
    await sslStream.AuthenticateAsServerAsync(serverCertificate,
       ...);
    return sslStream;
}
```

5.6.5 On the Server

GraphQL Endpoints

CSR Signing

On the server, the `SignCertificate` mutation receives a CSR associated with a device and returns the signed certificate string. The actual signing is performed by the `SignCsr` method in the `CAServer` class.

```
public async Task<string> SignCertificate(..., string csrPem)
{
    var device = await dbContext.Devices.
    FirstOrDefaultAsync(x => x.Id == deviceId);
    if (device == null) return null;
    string signedCert = LinkWaveServices.
    Utilities.Security.CAServer.SignCsr(csrPem);
    return signedCert;
}


public static string SignCsr(string csrPem)
{
    Pkcs10CertificationRequest csr = ParseCsr(csrPem);
    X509V3CertificateGenerator certGen = new
       X509V3CertificateGenerator();
    certGen.SetPublicKey(csr.GetPublicKey());
    certGen.AddExtension(X509Extensions.AuthorityKeyIdentifier,
       ...);
    certGen.AddExtension(X509Extensions.BasicConstraints, ...);
    X509Certificate signedCert = certGen.Generate(...);
    return ConvertToPem(signedCert);
}
```

Certificate Validation

On the server side, the `ValidateCertificate` method is used to verify the validity of a certificate.

The `ValidateCertificate` method takes a certificate as a PEM string and returns a boolean value indicating whether the certificate is valid.

```
public bool IsCertificateValid(string certPem)
{
    return LinkWaveServices.Utilities.
    Security.CA.ValidateCertificate(certPem);
}
```

6 Conclusion

With AirDrop, Apple revolutionized the interoperability of devices. Apple devices constantly exchange data in the background. When something is copied to the clipboard on one device, it can magically be pasted on another device. The variety of devices and operating systems, as well as Apple's monopolistic behavior, significantly hinder interoperability between Apple devices and those of other manufacturers.

LinkWave brings these features in the form of native apps to the four major platforms: Windows, macOS, Android, and iOS. Similar to AirDrop, files, texts, and images can be seamlessly exchanged between different devices. Through its local functionality, LinkWave makes data exchange much faster and more secure than existing cloud alternatives. Central to the platform-independent architecture is the ability to deliver a consistent and intuitive user experience, while ensuring high security standards through end-to-end encryption. The application leverages cutting-edge technologies like SwiftUI and Jetpack Compose to optimize integration and functionality across platforms.

Looking ahead, LinkWave could be further developed to support more device types and platforms, further enhancing the app's universal accessibility and usefulness. Future developments could focus on increasing data transfer efficiency and expanding the application with additional features that better meet users' needs.

List of Figures

1 iOS LinkWave App . . . 12
2 iOS Share Extension . . . 13
3 macOS LinkWave App . . . 14
4 macOS Share Extension . . . 15
5 macOS Menu Bar Icon . . . 15
6 Android LinkWave App . . . 16
7 Android Share Extension . . . 17
8 Selection Menu . . . 18
9 Windows LinkWave App . . . 19
10 Windows Share in Explorer . . . 20
11 Windows Share Window . . . 20
12 Windows System Tray Icon . . . 21
13 Windows System Tray Menu . . . 21
14 LinkWave Dashboard . . . 22
15 WinUI Logo . . . 25
16 SwiftUI Logo . . . 27
17 Jetpack Compose Logo . . . 29
18 Svelte Logo . . . 30
19 GraphQL Logo . . . 31
20 Bonjour Logo . . . 34
21 macOS .dmg Installer . . . 39
22 OS - Features . . . 46
23 UML of LinkWaveService . . . 48
24 MVVM Architecture . . . 49
25 Packaged App Installation . . . 51
26 Backend Architecture . . . 56
27 Device Discovery . . . 62
28 UML of LinkWaveDevice . . . 66
29 UML of BonjourController and BonjourAdvertiser . . . 68

30 Process of Device Discovery 68
31 UML of DiscoveryListener 70
32 Process of Advertising the Device 73
33 UML of RegistrationListener 74
34 UML Discovery 75
35 UML Advertisement 76
36 Devices Table 77
37 File Sharing 82
38 File Transfer Process on Android 87
39 UML of FileSendManager 90
40 UML of FileTransferOutgoing 91
41 File Transfer Process on Android 93
42 UML of RequestListener 94
43 UML of FileReceiveManager and FileActionReceiver 94
44 UML LinkWaveSender 97
45 Clipboard Sharing 104
46 UML of ClipboardShareRequest 108
47 Clipboard Sharing Process 109
48 Clipboard Sharing Process on Android 111
49 UML of ClipboardReceiveManager 112
50 Users Table 119
51 Structure of a JWT 122
52 Google Logo 124
53 OAUTH Flow 125
54 2FA Flow 128
55 LinkWaveService Class Diagram 145
56 LinkWaveService Notification 146
57 SSL/TLS 149
58 UML Certificate 158

List of Tables

1 Supported Features on Different Operating Systems 8

References

[1] WinUI Microsoft Documentation https://learn.microsoft.com/en-us/windows/apps/winui/

[2] Swift GraphQL https://swift-graphql.com/introduction

[3] KeychainSwift https://github.com/evgenyneu/keychain-swift

[4] Jetpack Compose https://developer.android.com/develop/ui/compose

[5] Apple Background Updates https://developer.apple.com/documentation/uikit/app_and_environment/scenes/preparing_your_ui_to_run_in_the_background/using_background_tasks_to_update_your_app

[6] Kotlin Flow https://kotlinlang.org/docs/flow.html

[7] Kotlin Serialization https://kotlinlang.org/docs/serialization.html

[8] TLS/SSL https://www.cloudflare.com/learning/ssl/transport-layer-security-tls/

[9] Chilli Cream https://chillicream.com/docs/hotchocolate/v13

[10] Microsoft SSLStream Documentation https://learn.microsoft.com/en-us/dotnet/api/system.net.security.sslstream?view=net-8.0

[11] Microsoft CSR Documentation https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.certificaterequest.createsigningrequest?view=net-8.0

[12] Microsoft X509Certificate2 Documentation https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.x509certificates.x509certificate2?view=net-8.0

[13] Google OAuth https://developers.google.com/identity/protocols/oauth2

[14] JWT https://jwt.io/introduction

[15] Two Factor Authentication https://learn.microsoft.com/de-de/aspnet/core/security/authentication/mfa?view=aspnetcore-8.0

[16] HOTP vs TOTP https://rublon.com/blog/hotp-totp-difference/

[17] The Bonjour Protocol https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/NetServices/Introduction.html

[18] GraphQL Request https://www.npmjs.com/package/graphql-request