Hes·so VALAIS WALLIS
Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

Degree Programme

Systems Engineering

Major Infotronics

BACHELOR'S THESIS

DIPLOMA 2023

Adrien Rey

NetPrinting Wireless Identification

- *Professor*
  Prof. Christopher Métrailler
- *Expert*
  M. Christophe Pierroz, Polyright SA
- *Submission date of the report*
  18.08.2023

swissuniversities EQUAL-SALARY CERTIFIED

Filière / Studiengang **SYND**	Année académique / *Studienjahr* **2022-23**	No TB / *Nr. BA* **IT/2023/86**
Mandant / *Auftraggeber* ☐ HES—SO Valais ☒ Industrie ☐ Etablissement partenaire *Partnerinstitution*	Etudiant / *Student* **Adrien Rey** Professeur / *Dozent* **Christopher Métrailler**	Lieu d'exécution / *Ausführungsort* ☒ HES—SO Valais ☒ Industrie ☐ Etablissement partenaire *Partnerinstitution*
Travail confidentiel / *vertrauliche Arbeit* ☐ oui / ja ☒ non / nein	Expert / *Experte* *(données complètes)* **Christophe Pierroz**, christophe.pierroz@polyright.com Promenade du Canal 83, 1950 Sion	

Titre / *Titel*

NetPrinting Identification

Description / *Beschreibung*

La société suisse Polyright fournit des systèmes centralisés d'identification, d'accès et de paiement via carte RFID. Les développements futurs sont orientés vers une digitalisation de la carte RFID. Les utilisateurs pourront également utiliser leur smartphone.

Ce travail de Bachelor a pour but de faire évoluer une solution existante basée sur une authentification RFID en ajoutant une communication sans-fil Bluetooth Low Energy (BLE) et un lecteur communicant avec un smartphone. Ce nouveau système d'identification devra être intégrer aux plateformes existantes de Polyright. Le PoC à développer permettra à un utilisateur de s'identifier de manière sécurisée sur une imprimante, copieur ou un appareil multifonction (MFP).

Objectifs / *Ziele*

— Analyse du système d'identification de Polyright existant et mise en place d'un banc de test pour l'identification d'un MFP. Sélection de lecteurs BLE disponibles.
— Spécification du protocole d'authentification sans-fil. Analyse des aspects de sécurité (authentification, replay attack, expiration des données, etc.).
— Remplacement du lecteur RFID ou ajout d'un module BLE. Programmation du lecteur et du protocole.
— Analyse des framework de développement mobile cross-platform existants (iOS, Android). Développement d'une application d'authentification de test sur smartphone, intégrée dans l'environnement Polyright.

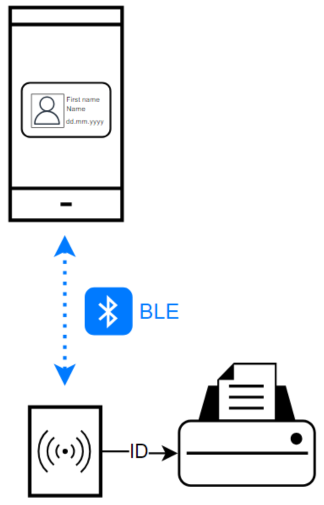
Signature ou visa / *Unterschrift oder Visum* Responsable de l'orientation / *Leiter der Vertiefungsrichtung:* [illegible] .... [1] Etudiant / Student: [illegible]	Délais / *Termine* Attribution du thème / *Ausgabe des Auftrags:* **15.05.2023** Présentation intermédiaire / *Zwischenpräsentation*: **19 - 20.06.2023** Remise du rapport final / *Abgabe des Schlussberichts:* **18.08.2023, 12:00** Expositions / *Ausstellungen der Diplomarbeiten:* **25.08.2023** – HEI **28.08.2023** – Monthey **31.08.2023** – Visp Défense orale / *Mündliche Verfechtung:* **Semaine/Woche 36 (04-07.09.2023)**

1 *Par sa signature, l'étudiant-e s'engage à respecter strictement la directive DI.1.2.02.07 liée au travail de diplôme.*
Durch seine Unterschrift verpflichtet sich der/die Student/in, sich an die Richtlinie DI.1.2.02.07 der Diplomarbeit zu halten.

Rapport reçu le / *Schlussbericht erhalten am* ……….………. Visa du secrétariat / *Visum des Sekretariats* …………..

Hes·so VALAIS WALLIS

Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften



NetPrinting Wireless Identification

Graduate Adrien Rey

Objectives

The company *Polyright* provides identification, access and payment using RFID cards. This thesis aims to implement a Bluetooth Low Energy (BLE) communication between a smartphone and a card reader to provide them an alternative to the cards.

Methods | Experiences | Results

For testing purposes, a test bench has been set up to replicate the *Polyright* ecosystem. It contains a Print Manager server connected to a printer. This allows printers, users and costs to be managed.

The middleware component connects the server to the mobile application. It also provides cryptographic methods. This component is a Spring web service coded in Java.

The mobile application has been developed using React Native. It can receive values from the server. The application connects to the card reader via BLE and authenticates itself. To achieve this, the card reader and the application use an authentication protocol based on AES-128 encryption.

The *Elatec* TWN4 card reader firmware has been adapted, using the provided API methods, to enable this authentication. This code was written in C.

This project has demonstrated that BLE communication between a smartphone and a card reader is possible in addition to existing RFID cards identification. Furthermore, it can be done in a secure way.

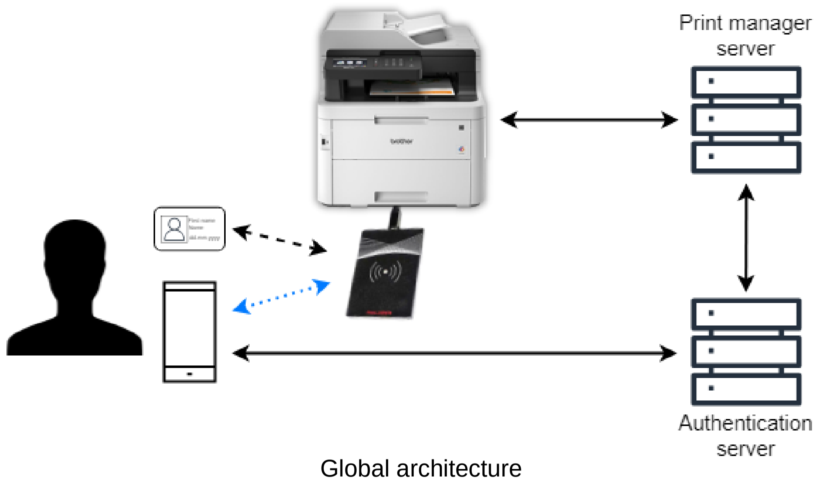
Bachelor's Thesis
| 2023 |

Degree programme
Systems Engineering

Field of application
Infotronics

Supervising professor
Prof. Christopher Métrailler
christopher.metrailler@hevs.ch

Partner
Polyright SA



Global architecture

swissuniversities EQUAL-SALARY CERTIFIED Valais excellence

Information about this report

Contact information

Author:	Adrien Rey Bachelor Student HES-SO//Valais Wallis Switzerland
Email:	adrien.rey@students.hevs.ch

Declaration of honor

I, undersigned, Adrien Rey, hereby declare that the work submitted is the result of a personal work. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and the author quotes were clearly mentioned.

Place, date: Sion, 18th August 2023

[illegible]

Signature: ______________________

Abstract

The Swiss company *Polyright* provides centralised identification, access and payment systems using RFID cards. Future developments will focus on the digitalisation of RFID cards. Users will be able to use their smartphone instead of a card. In most cases a QR code will be used. Both the smartphone and the device will be connected via a cloud. The presence of the phone would therefore be reported to the device using the latter.

However, this solution is not possible for printers. Each *Polyright's* customer uses the brand and model of printer he prefers. It would be necessary to adapt the program for each new device model to enable communication with the cloud.

This bachelor thesis aims to develop a new authentication solution:

- Based on a existing RFID identification solution.
- Using Bluetooth Low Energy (BLE) communication with a smartphone.

This new authentication system has to be secure to prevent any kind of attack.

Keywords: Bluethoot Low Energy, Authentication, RFID, Net printing

Contents

List of Figures

List of Tables

List of Listings

1 | Introduction

This thesis takes place in a world that is going digital. Digitalisation is a logical consequence of technological development and, in particular, of the Internet and IT. It is a process aimed at transforming traditional processes, objects and tools by means of digital technologies in order to make them more practical, easier to use and more efficient. This movement began long ago, with letters becoming emails, payments being made through banking portals, or music no longer requiring CDs to be listened to. However, the process has accelerated with the development and democratisation of smartphones and ever-increasing access to networks.

The smartphone is increasingly being used for all kinds of tasks: entertainment, social networking or shopping, for example. More and more valuable services are being offered on smartphones every day. From mobile payments to wallet apps for all kinds of cards or even train tickets. According to the Comparis.ch study *Etude smartphone 2020*, more than 97% of the Swiss population will use a smartphone in 2020. This number is increasing every year. In 2017, 7% of the Swiss population did not use a smartphone. These people are mostly over 54 years old. In the 18-35 age group, only 1 person out of 500 did not use a smartphone [1].

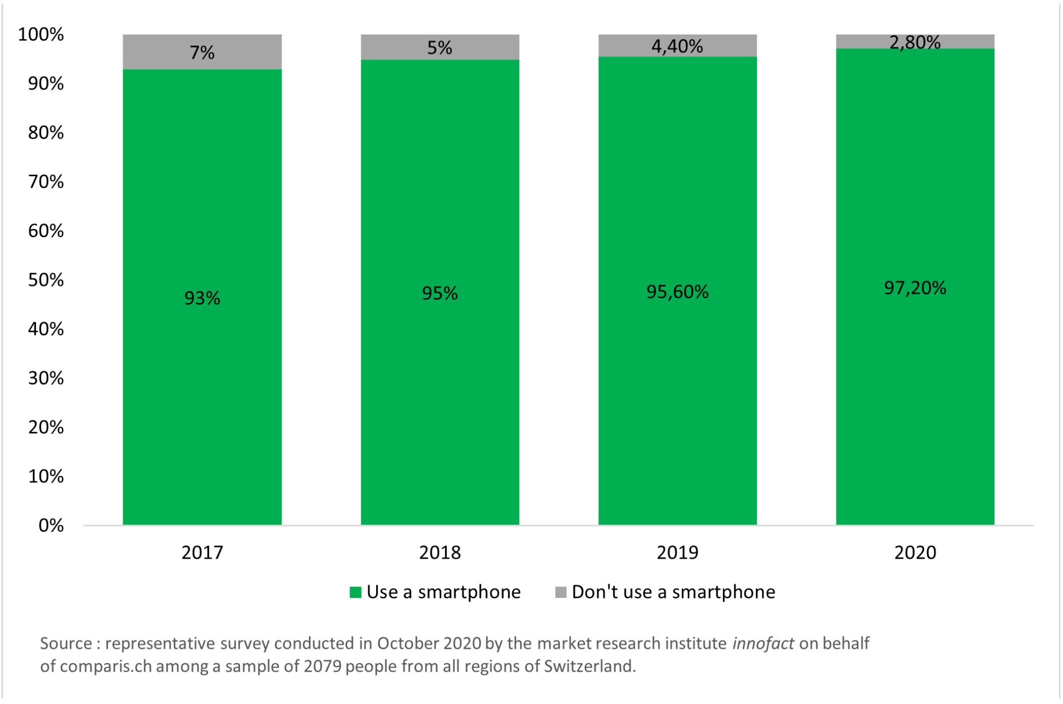


Figure 1.1: Smartphone users in Switzerland (Source: *Etude smartphone 2020.* ©Comparis.ch [1])

More and more people are using smartphones for more than just calling or texting. This observation is prompting many companies to start digitising in order to meet the needs and wishes of their customers. *Polyright*, the industrial partner in this project, has come to the same conclusion and has therefore focused its future developments on digitalisation. *Polyright* is a Swiss company that provides centralised identification, access and payment systems using Radio Frequency Identification (RFID) cards. Its future developments aim to offer its customers a digital card on a smartphone in addition to the RFID card. This digital version will have to perform the same tasks as the physical one.

Polyright is currently developing a digital payment solution. A mobile application for access control already exists and will be integrated into the *Polyright's* environment. However, they have questions about the identification process, particularly for the Multifunctional Printer (MFP) or Multifunctional Divice (MFD).[1] The current system allows to identify yourself to a printer using an identifier or a RFID card.

The aim is to be able to identify using a digital card on a smartphone. The proposed solution to this problem uses the Bluetooth of the user's smartphone via the *Polyright's* application to identify itself to the printer's card reader.

The application does not only transmit the user's ID, it has to provide a secure Bluetooth Low Energy (BLE) communication. All type of attack have to be avoid [2]:

- **Denial of Service** : The purpose of this attack is to overload the device with redundant packets, which make the device unusable.
- **Eavesdropping** : An attacker eavesdrops on a data exchange in order to extract useful information.
- **Man in the Middle** : A malicious device secretly establishes a connection between two devices and making them think they are exchanging data with each other.
- **Replay Attack** : A previously sent valid message captured by an intruder can be used to exploit the system functionality without an authentication procedure.
- **Relay Attack** : A malicious device establishes communication between two nodes and transmits unmodified data between them.

This involves setting up an authentication process. In computer security or access control, a distinction is made between identification and authentication. Identification establishes identity, while authentication verifies that the identity is genuine and belongs to the right person.

[1]**For ease of reading the term printer will be used in this paper to refer to printers, MFPs and MFDs.**

Objectives

This thesis is a Proof of Concept (PoC) that aims to develop a solution based on RFID authentication by adding BLE communication with a smartphone application. This new system will be used for NetPrinting identification.

Five main objectives can be defined for this thesis.

1. **Analysis of Polyright's existing identification system and ecosystem**: If the results of this work are convincing, it will be integrated into *Polyright's* platforms. The PoC must be as close to reality as possible to facilitate future implementation. To achieve this, the existing system must be analysed and understood.

2. **Setting up a test bed to simulate a real situation**: In order to develop and test NetPrinting Identification, a test bench must be set up to recreate a real-life situation. This test bench is a simplified version of the complete system.

3. **Analysis of the security aspects and specification of the authentication protocol**: The communication between the application and the card reader must be secured. An identification protocol must be implemented to avoid all kinds of attacks.

4. **Programming the RFID/BLE reader with the authentication protocol**: The firmware of the RFID card reader must be updated to accept BLE communication. It must integrate the authentication protocol.

5. **Analysis of existing cross-platform mobile application frameworks (iOs, Android) and development of an authenticated smartphone test application**: A smartphone application needs to be developed to perform BLE authentication with the card reader. This application can be implemented from an existing sample model.

An additional objective, defined by the school, was to analyse the impact of this thesis on sustainable development according to the United Nations Sustainable Development Goals. This analysis is available in the following appendix: Impact of the Bachelor Thesis on Sustainability.

Outline

Before presenting the project architecture, the industrial partner's ecosystem is analysed in the Chapter 2. The architecture of the project will be defined. Finally, some existing technology is presented.

Chapter 3 discusses the features and usage of the print management software.

Chapter 4 focuses on the middleware. It explains the choices for this part and the implementation of this service.

Chapter 5 deals with the implementation of Bluetooth Low Energy communication. The solution provided by the card reader manufacturer is analysed in detail. The authentication protocol is presented. The implentation of the card reader and the mobile application are detailed.

Chapter 6 concludes the report by analysing the results. Encountered problems and future improvements are presented.

All the code and documentation for this project can be found in the following repository: NetPrinting_Identification_Prog

2 | Analysis

Analyses are necessary to achieve the five objectives of this bachelor thesis. In order to carry out this type of work, it is necessary to have a good knowledge of the subject to be dealt with. This allows to think about all the different possible solutions or technologies. In addition, the analysis of existing solutions is important because it highlights similar work that can inspire or show some ways forward. Incidentally, the goal one is a analysis of the existing *Polyright's* ecosystem. The objectives 3 and 5 also include a analysis part.

If the results of this work are convincing, they will be incorporated into the solution offered by the industrial partner. The analysis of the existing *Polyright's* identification system should facilitate the future implementation of this Proof of Concept in their ecosystem. The PoC should strive for maximum fidelity to reality. The choice of technologies and implementations should always be made in this way. The industrial partner has provided an architecture for this project. It must be analysed and adapted to the needs and constraints of this project. These analyses will be developed in this chapter because they define the line of action of this thesis. Those included in objectives 3 and 5, as well as the choice of technologies, will be developed in the corresponding chapters. As different technologies are used, it has been decided to deal with each of them in its own chapter.

2.1 Polyright's ecosystem

The complete ecosystem will not be thoroughly analyzed. Their cash register system and access control will not be discussed. It is not relevant to this thesis. This analysis will focus on card identification and their NetPrinting systems.

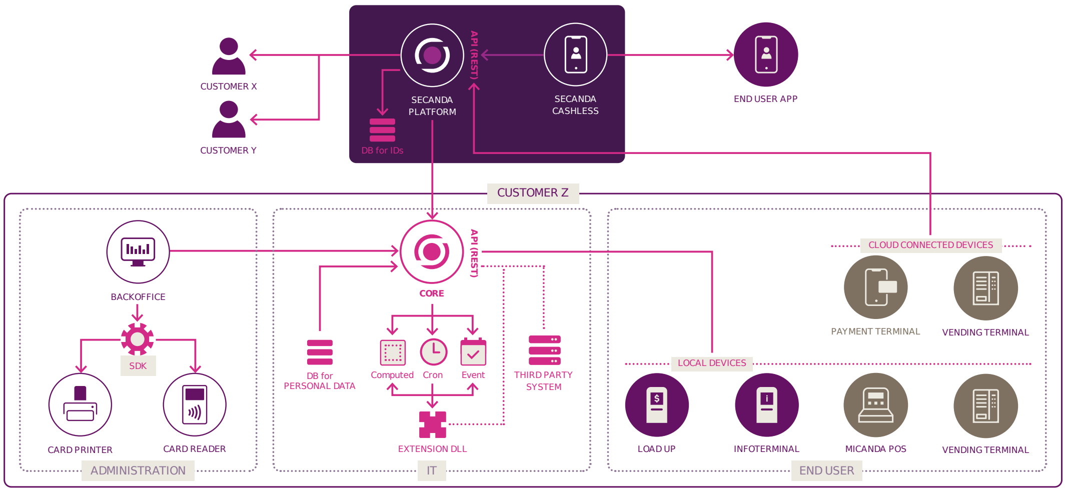


Figure 2.1: Architecture Secanda solution (Source: ©Polyright)

Polyright offers the *Secanda* solution to manage all the functionalities of an identification card. It also links all the services provided for the card. The solution is installed locally for each customer. It is divided into three main parts. Firstly, the administration has the *Secanda manager* tool to manage the service. It allows the creation and management of users and their cards, the control of the user's wallet and the printing service. Cards can also be read, printed or programmed with the manager using an external card reader or printer.

The front end of the Manager is an application for the user, called the back office in the figure below. It is the user interface of the *Secanda* solution. The backend is an application running on Azure or IIS, depending on the client. The backend accesses the local server with the API REST provided by *Secanda*. This server contains all the personal data of the customer's users.

All local devices, including RFID card identification, such as vending terminals or cash registers, are connected to the server via the API so that they can access the identified user's data. Third party systems are also connected to the server in the same way. The Print Manager is one of these third party services. The Print Manager is a service that monitors and controls printing tasks and resources. It is analysed and explained in more detail in the chapter 3. It is used in their NetPrinting product. It allows to manage printing and copying costs. Users can also print to printers and pay for their copies through their accounts.

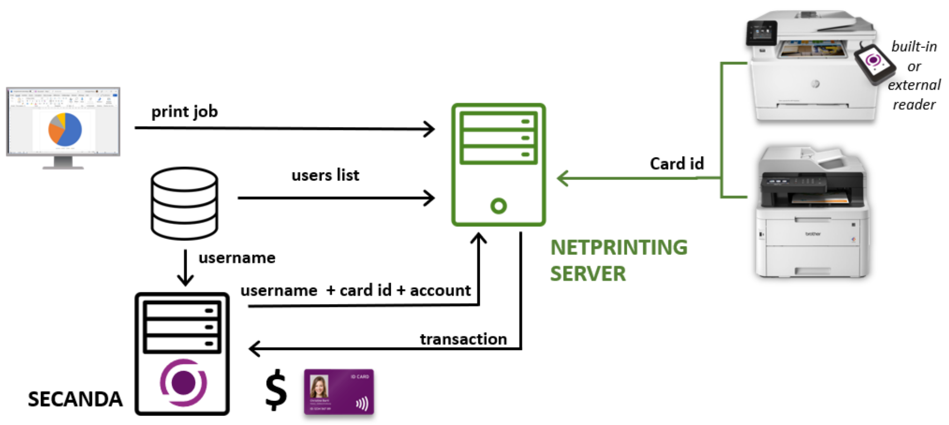


Figure 2.2: Polyright's NetPrinting architecture (*Source: ©Polyright*)

In the current *Polyright's* NetPrinting architecture, when a user sends a print job, it is stored on the NetPrinting Server (part of the Print Manager) until the user releases it at a printer. When the user identifies themselves with their card at the printer's card reader, the print manager checks the existence and balance of the user. Once these actions have been carried out, the user can release the job on the screen. The document is printed and the transaction is sent to *Secanda*. All the communication between the printer manager and the *Secanda* server is done with the API REST. All this can be done without identification or transaction, if the client so wishes.

Finally, *Secanda* provides a second API REST called *Platform*. [3]. This API can be used to access all customer systems. Access rights are limited to *Polyright's* developers and financial services. However, some cloud connected devices (mobile payment terminals or vending machine terminals) access the local customer server through this API. The customer's end user and the *Polyright's* mobile application also use this service to operate.

2.2 Technologies

As explained in the introduction, the industrial partner wants to offer a digital card with the same functionality as a physical card. The development of this new system involves several technologies. This thesis will focus on NetPrinting identification, but to understand the choices made, it is necessary to analyse the other technologies to be used. As part of their plans, they already have an existing mobile access control application that will be integrated into the *Polyright's* ecosystem in the future. They are currently working on the development of a digital payment solution. The planned payment solution (cash register or dispenser) is as follows:

1. Via *Polyright's* application, the user scans the payment device's QR code.
2. The payment device is informed via a cloud of the attempted payment.
3. The user is identified and the transaction can continue.

This QR code scanning solution could have been an excellent alternative to Bluetooth Low Energy identification. Unfortunately, this is not possible with NetPrinting. Most printing devices are connected to the network and can access a cloud for identification. However, each customer uses their own brand and model of printer. The solution would have to be customised for each type of printer. This would require a lot of work and could take time for the customer to add a new printer to their network.

The simplest technology would have been the Near Field Communication (NFC). All modern smartphones are equipped with it. NFC enables contactless communication between devices when they are brought within a few centimeters of each other. It is often used for mobile payments or data transfer. The NFC provides the ability to emulate a RFID card (Host Card Emulation (HCE)). For Android smartphones, the card can be emulated by a separate chip in the device, called a secure element. Android 4.4 and later provide an alternative card emulation method called host-based card emulation (HCE), which does not require a secure element. With HCE, any Android app can emulate an NFC card and communicate directly with NFC readers [4]. For iOS smartphones, it is possible to read and write data to tags and interact with protocol specific tags such as ISO 7816 or ISO 15693 tags. Unfortunately, there are no options to perform Host Card Emulation in iOS [5]. This makes it impossible to use this option to solve the printer identification problem.

Polyright's had to find another solution to achieve this identification. The new version of the reader card used (*Elatec TWN4*) contains a BLE module in addition to the RFID reader. This would allow the Bluetooth Low Energy to be used. It will enable communication between a smartphone and the card reader with the aim to identify. This technology was therefore chosen to solve the problem.

2.3 Polyright's BLE identification architecture

Once the technology was selected, the industrial partner considered an architecture for this solution that would fit into its ecosystem. It arrived at the following diagram.

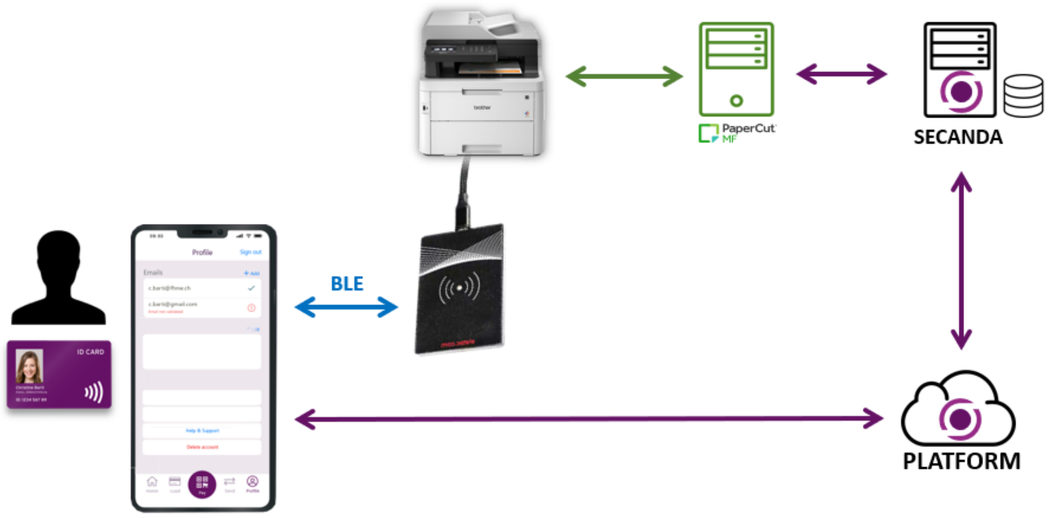


Figure 2.3: Polyright's BLE identification architecture (Source: ©Polyright)

The procedure for the printing part remains the same as the current NetPrinting solution, as explained in the section Polyright's ecosystem. The only difference is that the user identifies himself with his smartphone using BLE. The RFID card reader emulates a keyboard that sends the card ID read to the printer (most of them can read in text format). The printer will always behave in the same way, whether the identification is done by card or by smartphone. However, it is necessary to modify the reader's firmware to support Bluetooth Low Energy communication.

In this solution, the Bluetooth of the user's smartphone is used via the *Polyright's* application. It communicates with the card reader. The application is connected to the client's *Secanda* server using the *Platform* API in order to receive the user's data. The user must be controlled by the card reader using an authentication protocol to perform the authentication. This is analysed and defined in the chapter 5. If a message needs to be signed to validate the identity, this must be done on the server. Without this constraint, the mobile application could work locally without connecting to the server.

2.4 Bachelor thesis BLE identification architecture

The existing architecture is a good base, but it needs to be adapted to respond to the limitations of this project, be they temporal or technical. The main problem is the *Polyright's* ecosystem. It is a complex system and takes time to master. Moreover, setting up a test ecosystem is complicated. As the time available for this work is limited (14 weeks), priority is given to Bluetooth communication. The *Polyright's* ecosystem will not be used. This decision was taken in agreement with the professor and the industrial partner. The following architecture is therefore derived from the above and the choices made.

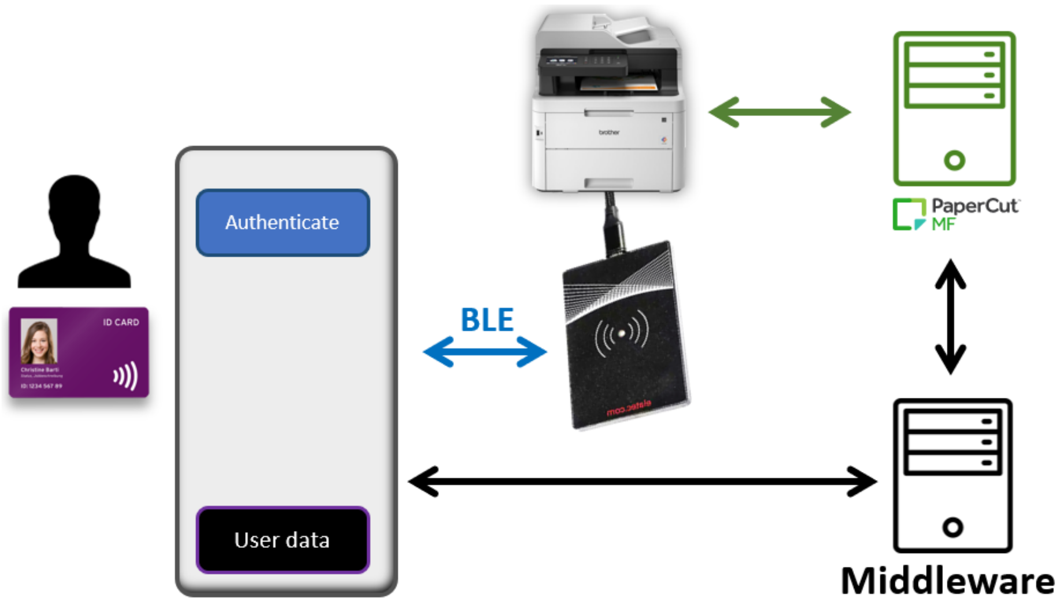


Figure 2.4: Bachelor's thesis BLE identification architecture

In this architecture, a simple application must be developed instead of the *Polyright's* application. It must perform authentication via Bluetooth Low Energy. The smartphone will communicate with the **Middleware** service. The latter must also be developed and will replace the *Secanda* server and the *Platform* API. The **Middleware** must allow interaction with the print manager and provide the necessary methods for the authentication process. The technology choices and implementations are detailed in the following chapters Middleware and BLE communication. All of these choices have been made with a view to remaining as faithful as possible to the ecosystem and facilitating future implementation in it.

The print manager used in this project is provided by the industrial partner, as explained in the chapter Print Manager. The card reader is also provided. It is a *Elatec TWN4 Slim RFID/BLE card reader*. This model is used by *Polyright* for these customers. Finally, the printer for the test bench is provided by the school's IT department for the duration of the project.

2.5 Existing solutions

This section presents and discusses existing or similar solutions. No projects or examples of Bluetooth communication between a smartphone and a *Elatec* card reader (TWN4) were found. Nevertheless, the manufacturer provides two services to perform authentication.

The first solution is *KleverKey*. It is a software-as-a-service product that turns smartphones into virtual keys [6]. *KleverKey* enhances card readers by providing mobile authentication functionality. It is integrated into *Elatec's* development package and requires straightforward configuration, as simple as adding a regular RFID tag. This solution is dedicated to access control. It also uses proprietary locks. It is not a basis or example for this project as no code or diagram is available.

Safetrust is the second solution. The Safetrust wallet for iOS and Android holds a secure virtual identity that provides contactless access to premises, resources and more. It provides a way to authenticate for secure printing, desktop login or secure access. This solution also uses proprietary products and does not provide access to their technology.

Elatec also provides a demo application to perform BLE authentication with the card reader. It can be downloaded from the App Store or Play Store. *Mobile Badge BLE NFC* transmits the phone's identifier to the reader. The relevant firmware image and installation instructions are available in the development pack. The source code is not accessible and cannot be analysed. However, the communication can be analysed with a Bluetooth sniffer. This will serve as a basis for implementing the communication and defining the authentication protocol. This analysis can be found in the 5 chapter.

Simple firmware examples are also included in the TWN4 development package. This is the basis for implementing firmware that supports BLE. The print manager also comes with many code samples, making it easy to integrate into the system. The middleware is a simple service, and the example provided in the manual is sufficient [7].

Finally, there are plenty similar solutions for the mobile application on GitHub. The following was chosen to inspire the architecture of the mobile application: **BluetoothLowEnergySample** from *friyiajr* [8]. It discovers, connects and streams data between a mobile application and a device.

3 | Print manager

Although computers, tablets and smartphones are becoming more and more common, paper is still an essential part of daily lives, even more so in a university or high school where students need a lot of documents. The digitalisation of course materials and exercises is increasingly being encouraged, even in smaller courses, through platforms such as *Moodle*. However, not everyone likes to work with a tablet or a virtual document. Some people prefer to work with printed documents. There are also some activities that cannot be done on a computer. Examinations are a good example, although ways are being developed to allow examinations to be taken on a computer. This trend towards digitisation is not only because of the increasing use of electronic devices, but also for environmental reasons. Society wants to preserve our planet and one goal is to reduce the use of paper. To achieve this goal, society wants to avoid the unnecessary use of paper. An excellent way to reduce the number of printed documents in a school or company is to make paid prints. The user has a monthly or annual print credit and when it is used up they have to pay for their own printing.

In this context, the use of a print manager makes sense. A tool is needed to manage these credits, impressions and printers on a large scale, and print managers meet these needs. They allow the creation of users, with or without credits, to manage the fleet of printers and the necessary drivers. Print managers also help to control the impressions through reports or dashboards. There are many solutions on the market. Some are developed by printer manufacturers and are specific to their devices, such as *uniFLOW* for *Canon* devices or *XEROX Print Management*. Others are designed to be universal and work with all types of printers. This type of solution is interesting for large companies or schools because it allows them to have different printer models or brands. The school where this bachelor's thesis takes place has chosen the *PaperCut MF* print manager to manage its printouts.

PaperCut is a company that provides print software solutions such as cloud print management or print management servers. This thesis will focus on the print management software called *PaperCut MF*. The other services offered will not be discussed.

The industrial partner of the bachelor thesis has proposed this tool to manage the prints as it was used in the school. They are also familiar with this service as they use it regularly. It was therefore decided to keep this service for the architecture of the project. This allows us to stay close to the proposed solution. In addition, a development license was granted for the duration of the project.

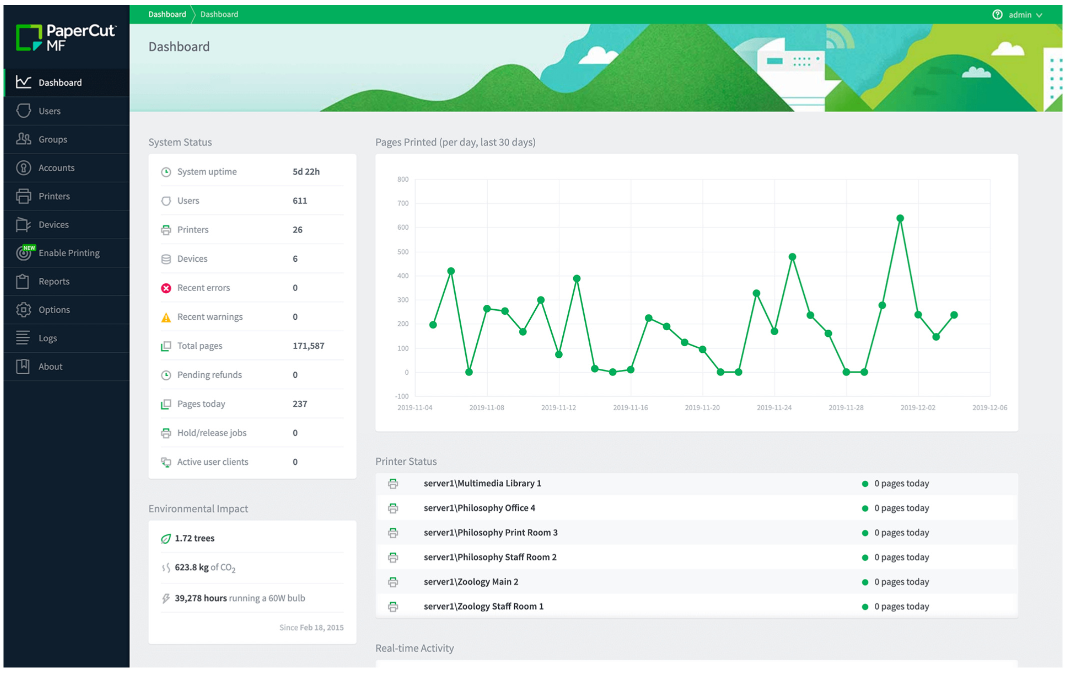
3.1 Features

PaperCut MF is a server that enables centralized management of every user and printing device installed on Windows, iOS, or Linux (x64). Many features are available:

- Track and control all print, copy, fax, and scan activity on printers.
- Manage user access rights to devices based on function.
- Implement copy quotas, charges, and per-page costs.
- Enhance document security with Find-Me print release at printers.
- Enhance device security with managed device access via user ID or building access cards.

This work is linked to these last two features. With Find-Me Print, users no longer need to select a specific printer for their print jobs. Instead, they can send their job to the shared Find-Me queue (a virtual printer). They can choose any device they prefer to authenticate and securely release their job. The manager provides user authentication to the printer with several login options. A unique ID, a username and password or a badge can be used. All of this data is stored on the server and can be changed manually from the dashboard or synchronised from sources such as Windows Active Directory, Azure Active Directory or LDAP [9].

A web admin dashboard is available to control and modify all server parameters, users or printers. It is accessible from any network location with the admin login.



System Status

System uptime	5d 22h
Users	611
Printers	26
Devices	6
Recent errors	0
Recent warnings	0
Total pages	171,587
Pending refunds	0
Pages today	237
Hold/release jobs	0
Active user clients	0

Environmental Impact

1.72 trees
623.8 kg of CO_2
39,278 hours running a 60W bulb
Since Feb 18, 2015

Printer Status

server1\Multimedia Library 1	● 0 pages today
server1\Philosophy Office 4	● 0 pages today
server1\Philosophy Print Room 3	● 0 pages today
server1\Philosophy Staff Room 2	● 0 pages today
server1\Zoology Main 2	● 0 pages today
server1\Zoology Staff Room 1	● 0 pages today

Figure 3.1: Dashboard PaperCut (Source: ©PaperCut)

A web service API is provided by *PaperCut* to interact with the server. All data is transferred over standard HTTP or HTTPS using standardised XML markup. This allows PaperCut to be integrated into the develop components. A complete list of all methods is available [10]. The rest of this document only describes the methods used.

They can also be called from the command line or automated using the server command. The latter is a command line tool that takes the commands as arguments and prints the results of the command to the console [11].

Some printers have a touch screen to control the device. *PaperCut MF* provides software for these displays. If a printer does not have a screen, the *PaperCut Print Release Station* can be used. It is typically used in full-screen mode on dedicated workstations near the printers [12]. This tool can be opened on the Virtual Machine (VM) or on the host. An image based on Rasbian GNU/Linux is provided by the vendor. The *PaperCut Print Release Station* can thus be set up on a Raspberry PI. This is a good alternative to not using a PC for publishing. This solution is closer to a real installation.

3.2 Setting up

According to the architecture chosen for this thesis, a *PaperCut MF* server has to be set up. It is installed on a Windows Virtual Machine to simulate a real server on another machine. The VM has been set up on *VM Ware*. It is a standard Windows 10 x64 machine. The network interface is set up as NAT. The virtual machine shared the IP address of the host.

The **quick Windows install** guide was then followed for the server installation [13]. The server is listening on port 9191. The user web interface is accessible at this address: *http://papercutserver:9191/admin* or *http://papercutserver:9191/user*. The admin account can then connect to the admin web interface to set up and modify the server. Standard costs have been defined. These costs are arbitrary and are only set up to see the print activity on a user credit.

For the first test, the session user was used to print documents. No printer was available, so a PDF printer was used to test the setup. To allow the server to see the virtual printer, line 402 of the *print-provider.conf* file (located in the VM: *C:/Program Files/PaperCut MF/providers/print/win/*) must be replaced with *IgnoreVirtualPrinters=off*. Virtual printers are no longer ignored and documents can be printed.

3.2.1 Print release station

For further testing, secure printing has been added to the printer settings. It requires user approval to print the document. The *Print Release Station* is used for this purpose. It can be opened by running the *pc-release* application. The latter can be accessed either from the virtual machine or from the host using the shared network. The executable is located in *C:/Program Files/PaperCut MF/release/* on the VM or *//papercutserver/PCRelease*. Before it can use, it need to be configure it. The station can be customised in the *config.properties* file. Line 127 has been replaced with *use-card-authentication=Y* to allow card authentication on the station. The *connection.properties* allows to set the server information for the host to connect to. The IP address of the server can be obtained by running the following command *ping -4 papercutserver* on a prompt. The name is the same as the shared network.

A sample image can be loaded into the card reader to perform a simple card detection. Identification can then be performed with the card reader connected. An old RFID card was used. Its identifier was discovered with a RFID application. It must be added to the session user profile under the primary ID number. A print job can now be released with the *Print Release Station* using the card reader. This validates the behaviour of the print manager.

The release station can also be set up on a Raspberry Pi. This has been tested using a Raspberry Pi Model 3 with a 7" touchscreen display. The provided image can be flashed to an SD card (minimum 16GB). The server properties can be defined in the *pc-connection.properties* file. As an application, the station image can be customised in the *pc-config.properties* file. Once mounted, the Raspberry will boot and start the release station. It will attempt to connect to the server. The Raspberry Pi can be accessed using an ssh connection (hostname: pcrelease). The default username is *admin* and the password is *password*. During this project, the print release station never managed to connect to the server. Various setups have been used to try to get access to the latter. A fixed IP address, on the same subnet, was configured for the host laptop and the Raspberry Pi. The address was changed in the control panel for the computer and by adding the following code with the desired address to the *dhcpcd.conf* file.

```
interface eth0
static_routers=192.168.4.1
static domain_name_servers=192.168.1.1
static ip_address=192.168.4.30/24
```

Listing 3.1: Code - Example configuration server DHCP

This solution did not solve the connection problem. Tried to set up a DHCP server on the Raspberry. It always has a static address. The dnsmasq package can be downloaded (*sudo apt install dnsmasq*). In the *dnsmasq.conf* file, the DHCP configuration can be added like this.

```
interface=eth0
bind-dynamic
domain-needed
bogus-priv
dhcp-range=192.168.4.100,192.168.4.200,255.255.255.0,12h
dhcp-host=00:11:22:33:44:55,192.168.4.50
```

Listing 3.2: Code - Example server configuration DHCP

This line *dhcp-host* tells DNSMasq to assign a specific IP address to a device with the given MAC address, in this case the host computer. This solution didn't solve the problem either. The virtual machine is configured in NAT mode and therefore has the same address as the host. Bridged mode was also tried. The VM is connected directly to the physical network. Both static IP and DHCP failed to connect. The Raspberry PI implementation was abandoned to concentrate on the Bluetooth Low Energy part. In addition, the release application is functional on the machine, allowing tests to be carried out.

3.2.2 Print queue

If it has more than one printer, it is best to create a null port. This acts as a print queue. It appears to users as a printer and they can always use it. All print jobs can then be redirected to physical printers. The port was created following the **Setup a Nul Port** guide [14].

A *HP LaserJet P3015* printer was provided for this thesis. A cold reset had to be performed before the printer's IP address could be changed. When the printer starts up and memory initialisation begins, press the OK button until the three LEDs on the control panel are lit. The Cold Rest option can then be selected and executed. After the reset, a manual IP address has been set in the settings on the same subnet as the host computer. The Printer Pilot can be downloaded from `support.hp.com` (HP Universal Print Driver for Windows PLC6). Once the driver is running on the virtual machine, the printer can be added by searching for it by its IP address and following the configuration steps. The printer will then appear in the printer list on the server. Jobs can be added to this device. They can be copied from another printer. Finally, the null port can be redirect to this printer. This can be done in the Nul Port settings.

When a document is printed in the queue, it waits to be released by a card or the application. The document will then be printed on the physical printer. Users only see and print to the print queue. The physical printer is not visible to users.

3.2.3 PC client

Finally, an application can be opened on the host to select the user when printing a document. For this thesis, this will allow printing with different users using the same computer. On the print manager network, the *pc-client* application can be accessed under */PCClient/win*. The server IP address can be modified in the *config.properties* file. Line 144 needs to be changed to *default-selection=print-as-user* to enable the print-as-user functionality. Setup is complete and are ready to test the server.

3.3 PaperCut manager

The goal of the VM is to run the server in the background. In a real installation, the server is not necessarily accessible physically. It is necessary to be able to make changes to the server quickly to simplify future testing. As explained in the section 3.1, a XML API or a command line tool is provided. One of this technology will be used to set up communication between this host machine and the server. The objective is to be able to add a user or a list of users, delete a user or all and modify a user credit.

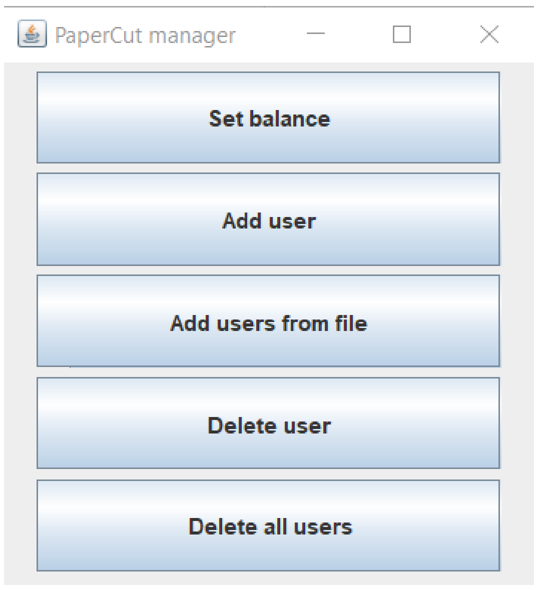


Figure 3.2: PaperCut manager

All the tools and libraries used for this component are available in appendix B.

3.3.1 Choice of the technology

Initially, a simple batch file was used to send command line requests. It works, but the problem is that it requires many lines of code for basic statements as a for loop. For example, the following code is the function to add a new user with a name, a primary card number, a secondary card number and a balance (the full code is available in the PaperCut manager folder in the repository).

```
echo Add new user
set /p name= Enter user name :
set /p pCardN= Enter primary card number :
set /p sCardN= Enter secondary card number :
set /p balance= Enter balance :
server-command add-new-user %name%
server-command set-user-property %name% balance %balance%
server-command set-user-property %name% primary-card-number %pCardN%
server-command set-user-property %name% secondary-card-number %sCardN%
```

Listing 3.3: Code - Example batch file

This batch file worked well on the virtual machine, but the goal is to change properties from the host machine. To achieve this, *PsExec* was used. It allows programs to be executed on remote systems. This solution was suggested by PaperCut in addition to this server command. The TCP port 445 and the UDP port 134 must be open to incoming traffic on the VM to allow communication with *PsExec*. This solution works well with a simple command prompt, but is slow. The connection has to be set up, and just to add a person it needs four commands (one for each parameter). Simple actions take time.

To avoid these problems, it was decided to move to a Java program using the XML API instead of the command line tool. The choice of Java for this application development was based on familiarity with the language. It also has the *Swing* graphics library which is part of the Java Foundation Classes (JFC) package. It allows the quick creation of a small graphical interface.

3.3.2 Implementation

The UML diagram below shows the architecture of the application. This application contains the main ServerCommandProxy class. It provides several methods [10]. This class is provided in the server file at *C:/Program Files/PaperCut MF/server/examples/webservices/java*. It creates a proxy for sending XML-RPC calls. It needs three parameters to create the proxy: the server name, the port and the authentication token. This last token can be modified according to the security procedure for the *PaperCut* web service [10]. This class requires the Apache XML-RPC Library version 2.0.1 and the Commons Codec Library version 1.3.1. This library can be obtained from the application server installation directory or online and added to the project libraries. The figure bellow shows the class diagram of this component.

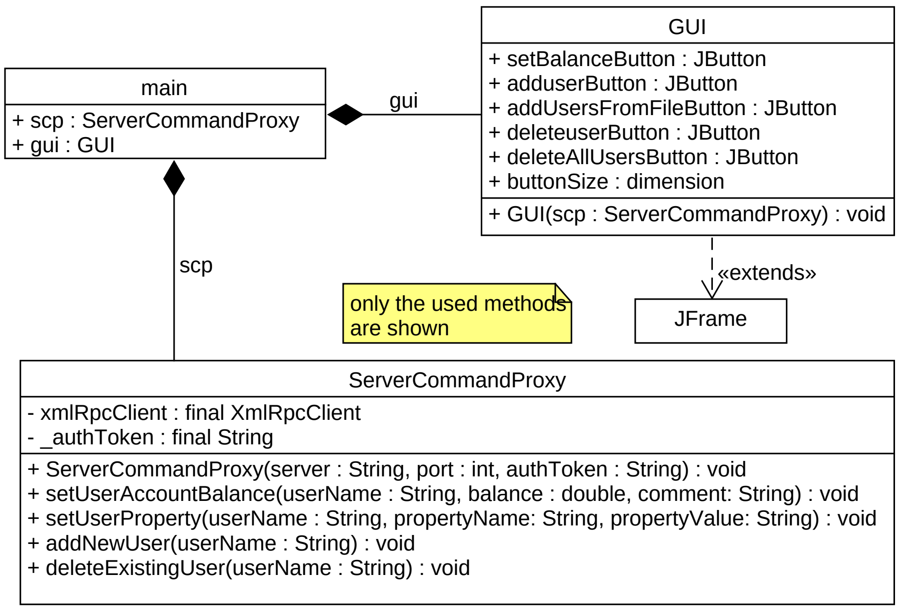

GUI
+ setBalanceButton : JButton
+ adduserButton : JButton
+ addUsersFromFileButton : JButton
+ deleteuserButton : JButton
+ deleteAllUsersButton : JButton
+ buttonSize : dimension
+ GUI(scp : ServerCommandProxy) : void
main
+ scp : ServerCommandProxy
+ gui : GUI
gui
scp
«extends»
JFrame
only the used methods are shown
ServerCommandProxy
- xmlRpcClient : final XmlRpcClient
- _authToken : final String
+ ServerCommandProxy(server : String, port : int, authToken : String) : void
+ setUserAccountBalance(userName : String, balance : double, comment: String) : void
+ setUserProperty(userName : String, propertyName: String, propertyValue: String) : void
+ addNewUser(userName : String) : void
+ deleteExistingUser(userName : String) : void

Figure 3.3: Diagram class PaperCut manager

A GUI class is also used to generate the user interface and logic. Five buttons have been created for the features. An action listener was implemented for each button to define its action. When a button is clicked, it asks for different parameters or confirmation in the following order and performs the requested action.

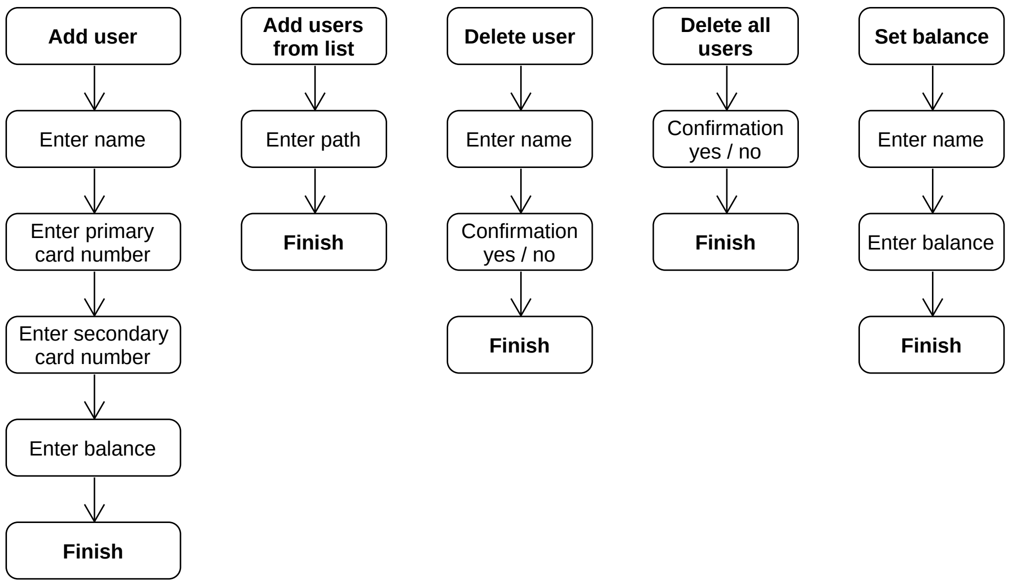


Figure 3.4: Sequence PaperCut manager

Different security is implemented to avoid a bad request. It checks that the name parameter is not empty or exists in *PaperCut*. It also checks that the balance parameter is not empty. If it is, the balance is set to zero. Other parameters are not critical, they can be used with a null value. A jar file has been created to make it easier to use the manager.

4 | Middleware

In the complete solution architecture, *Secanda* and the *Platform API* [3] allow to interact with the server and to connect all the services. The defined architecture replaces them, as explained in the 2. With this simplification, it becomes a client-server architecture between the mobile application and the *PaperCut MF* server. This is where the middleware comes in.

Middleware is software that facilitates communication and connectivity between different applications or components within a distributed network. It makes it easier to connect applications that were not originally designed to interact. It enables more efficient application development and faster time to market by providing intelligent integration capabilities. Middleware enables applications to be built without the need to build custom integration whenever they need to connect to application components, services, data sources or devices. Instead of starting from scratch, it can leverage a standardised and reusable set of tools and functionality provided by middleware [15].

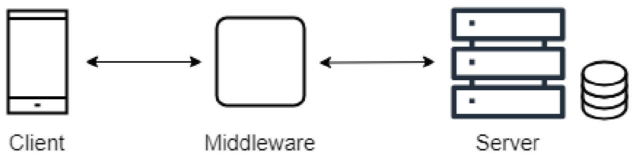


Figure 4.1: Client-server architecture with Middleware

Adding this component between the server and the mobile application was optional. The latter could communicate directly with the server. The use of a middleware component is a best practice in this case. It provides a standardised and reusable set of tools and functionalities, as explained. Also, since Bluetooth Low Energy communication must be secure, authentication is performed using encrypted data. This is not done in the mobile application. The application must have as little data as possible, such as encryption keys or user data, to avoid security problems. In the real environment, the data is stored in a secure database that can be accessed using the *Platform API*. In addition, it will allow the application to access data stored on the server. In particular, it will be able to retrieve the list of users, their ID or even their balance.

This Proof of Concept will run on a local machine, no security will be implemented between these three components. In the future, this middleware component will be replaced by the *Polyright's* technologies, which contain their own security mechanisms.

All the tools and libraries used to develop this part can be found in the appendix B.

4.1 Choice of the technology

Before choosing a technology, it was necessary to decide what form this component would take. The main objective of this thesis is to validate a BLE communication and not to create a complex component to interface between the server and the user. It was therefore decided to create a simple web service that communicates via HTTP. This choice was made because a simple web service is easy and quick to set up. In addition, the school and the partner company create and work with this type of service on a regular basis. They are therefore familiar with this type of service.

For the technology we were offered two options. On the one hand, to program a Java standalone application to run this web service, using a framework such as Spring boot or Vert.x. This solution is favoured by the school because Java is a language taught and regularly used for projects or laboratories. Moreover, these frameworks are well known. On the other hand, the web service is realised with .Net 6 and hosted on IIS. This solution is used by the partner company *Polyright* to implement and use its own web service.

A Java application based on the Spring boot framework was chosen because it is a language and a framework that the school is familiar with. Moreover, if this project were to be taken over in the future to implement the BLE communication, this web service would not be considered. It will be replaced by its environment. This led to the choice of a technology used and known by the institution.

This service will communicate on the one hand with the *PaperCut MF* server and on the other hand with the mobile application. For the server-side communication, a XML API is provided by *PaperCut*. For the user side, a API REST is implemented to allow operations to be performed via HTTP methods. This API was chosen because it meets the needs of the web service and is easy to use. SOAP could have been used, but it comes with more limitations as it is a protocol with requirements such as XML messaging. This choice was also made because *Polyright* uses API REST in its ecosystem. In this project, simple data is sent in JSON format with a label and a value. These values could have been returned in clear text, but this is limiting as the project evolves and more complex data is exchanged.

For the security part, the AES (Advanced Encryption Standard) algorithm is implemented. It can be used with 128, 192 or 256 bit keys. In this project a 128 bit key is used. This choice was dictated by the model of the card reader. In fact, it has a security module implemented by the manufacturer using CBC AES-128 without padding for the encryption. This procedure is sufficient for this project, the aim of the security part is only to authenticate the users with an authentication procedure using the AES-128 algorithm. According to the **Recommendation for Key Management** published by the National Institute of Standarts and Technology (NIST), a 128-bit key is considered secure and approved for use until 2030 [16]. This ensures a certain lifespan for the project, taking into account future advances in cryptanalysis.

To ensure correct encryption, the choice of key is really important. In most cases, the key is a generated random number. Not all random number generators are secure. Some random number generators produce completely predictable random numbers.

The shared key is set once when the system is installed. It is important that the key chosen is as random as possible. Furthermore, if the method of generation is unknown to the attacker (if it is not implemented in the firmware of the card reader or middleware), this makes the key even more secure[17].

4.2 Implementation

To start implementing the middleware, the **Building an Application with Spring Boot** guide was followed [7]. In the *Spring Initializr*, the software project manager can be selected. For this project, Maven was chosen. So it had to be installed to continue the process. The guide makes it possible to quickly set up a controller that returns a Hello World message with the following query *http://localhost:8080/* to a search engine.

By default the server listens on port 8080. This can be changed with a *application.properties* file containing this line *server.port=xxxx*, or by customising the server configuration directly in the program. Before proceeding with the implementation of the REST service, communication via the *PaperCut* XML API.

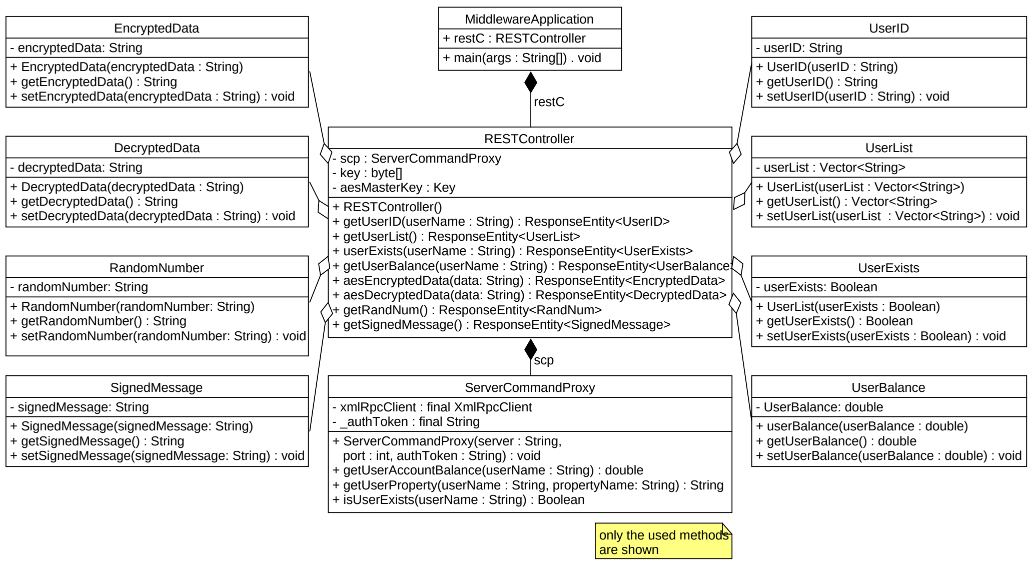

EncryptedData
- encryptedData: String
+ EncryptedData(encryptedData : String)
+ getEncryptedData() : String
+ setEncryptedData(encryptedData : String) : void
DecryptedData
- decryptedData: String
+ DecryptedData(decryptedData : String)
+ getDecryptedData() : String
+ setDecryptedData(decryptedData : String) : void
RandomNumber
- randomNumber: String
+ RandomNumber(randomNumber: String)
+ getRandomNumber() : String
+ setRandomNumber(randomNumber: String) : void
SignedMessage
- signedMessage: String
+ SignedMessage(signedMessage: String)
+ getSignedMessage() : String
+ setSignedMessage(signedMessage: String) : void
MiddlewareApplication
+ restC : RESTController
+ main(args : String[]) . void
restC
RESTController
- scp : ServerCommandProxy
- key : byte[]
- aesMasterKey : Key
+ RESTController()
+ getUserID(userName : String) : ResponseEntity<UserID>
+ getUserList() : ResponseEntity<UserList>
+ userExists(userName : String) : ResponseEntity<UserExists>
+ getUserBalance(userName : String) : ResponseEntity<UserBalance>
+ aesEncryptedData(data: String) : ResponseEntity<EncryptedData>
+ aesDecryptedData(data: String) : ResponseEntity<DecryptedData>
+ getRandNum() : ResponseEntity<RandNum>
+ getSignedMessage() : ResponseEntity<SignedMessage>
scp
ServerCommandProxy
- xmlRpcClient : final XmlRpcClient
- _authToken : final String
+ ServerCommandProxy(server : String, port : int, authToken : String) : void
+ getUserAccountBalance(userName : String) : double
+ getUserProperty(userName : String, propertyName: String) : String
+ isUserExists(userName : String) : Boolean
only the used methods are shown
UserID
- userID: String
+ UserID(userID : String)
+ getUserID() : String
+ setUserID(userID : String) : void
UserList
- userList : Vector<String>
+ UserList(userList : Vector<String>)
+ getUserList() : Vector<String>
+ setUserList(userList : Vector<String>) : void
UserExists
- userExists: Boolean
+ UserList(userExists : Boolean)
+ getUserExists() : Boolean
+ setUserExists(userExists : Boolean) : void
UserBalance
- UserBalance: double
+ userBalance(userBalance : double)
+ getUserBalance() : double
+ setUserBalance(userBalance : double) : void

Figure 4.2: Diagram UML Middleware

4.2.1 Print manager server communication

The proxy Java class *ServerCommandProxy* provided by *PaperCut* was then added (located on the VM at *C:/Program Files/PaperCut MF/server /examples/webservices/java*). This class is designed to wrap XML-RPC and calls XML API commands. It therefore requires the Apache XML-RPC library version 2.0.x (not a later version). This library can be obtained from the *PaperCut* server installation directory or online. The Apache Commons Codec Library version 1.3 must also be installed to encode and decode various formats such as Base64 and Hexadecimal. The jar file must be added to the Spring boot project folder, and the libraries must be added to the project libraries.

After these settings the class is almost ready to use. The *ServerCommandProxy* must be created with the following parameters:

- **server**: The name or IP address of the server hosting the application server. In this project the server name is *papercutserver*.
- **port**: The port on which the application server listens. The default port, 9191, is used in this project.
- **authToken**: The authentication token string. All RPC calls must pass an authentication token. For this application it is *authToken*. The authentication token must be defined in the *PaperCut MF* server parameters according to the security procedure in the [10] manual.

Both communications are set up. The REST functions are ready to be implemented.

4.2.2 Security class

Before going on to implement the REST service, a security class must be implemented to allow cryptographic operations to be performed. It can encrypt and decrypt data.

The constructor initialises the Cipher object with the correct transformation. The key is also generated with a hardcoded value. There are better ways to do this. One way is to generate a key when the Security object is created. This can be done by adding a generateAESKey function to replace the SecretKeySpec method.

```java
/**
* Generates a random AES encryption key with a key size of 128
    ↪ bits.
*
* @return The generated AES encryption key, or null if an error
    ↪ occurs.
*/
private Key generateAESKey() {
    try {
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        SecureRandom secureRandom = new SecureRandom();
        keyGen.init(128, secureRandom);
        return keyGen.generateKey();
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
        return null;
    }
}
```

Listing 4.1: Code - Example generateAESKey function

This function generates and returns a AES compatible 128 bit key. It will be different each time a new security object is created. The key is no longer hard coded. This solution is difficult to implement in this thesis. The key is shared between the card reader and the middleware. This would require changing the firmware of the card reader every time the web service is restarted. Furthermore, this component is tested locally and will be replaced by the *Polyright's* environment if it uses this Proof of Concept. This solution was therefore chosen.

In the encryptData and decryptData functions, the cipher object is initialised to the appropriate mode. The parameter is a hexadecimal number in the form of a string. The received string is converted into a bit array using the *decodeHex* method of the *Hex* Java class. Once encrypted or decrypted, the value is returned as a *Hex* string.

4.2.3 REST service

Below, the REST four types of requests are detailed. This will help to determine the needs.

- **GET**: The GET method retrieves a representation of a resource or collection of resources from the server.
- **POST**: The POST method submits data to the server to create a new resource. It sends data in the request body and the server processes it accordingly.
- **PUT**: The PUT method updates an existing resource on the server. It replaces the entire resource representation with the new data provided in the request body. If the resource does not exist, PUT can create a new resource with the given identifier.
- **DELETE**: The DELETE method deletes a given resource from the server. It instructs the server to remove the resource identified by the given URL.

For this web service, the REST client needs to retrieve information from the *PaperCut* server: the list of users, their identifier, their balance and whether a user exists. The GET method meets these needs, as it allows resources to be retrieved. The *REST-Controller* class is a Spring controller that uses the *@RestController* annotation. This annotation simplifies the creation of RESTful web services by eliminating the need to annotate each request-handling method of the controller class with a response body annotation. It allows us to write a method in the following form :

```
 1 /**
 2 * Get request for user name
 3 *
 4 * Handles a GET request to fetch a user ID based on their username.
 5 * Return it in JSON format
 6 *
 7 * @param userName userName to get userID
 8 * @return ResponseEntity containing the user ID
 9 */
10 @RequestMapping(method = RequestMethod.GET, path ="/getUserID")
11 public ResponseEntity<UserID> getUserID(@RequestParam String userName) {
12     UserID response = new UserID(scp.getUserProperty(userName,
13                                  "secondary-card-number"));
14     return ResponseEntity.ok(response);
15 }
```

Listing 4.2: Code - Example GET method

The *@RequestMapping* annotation on line 10 is used to map web requests to controller methods. It takes as arguments the method and path to throw that request. When a *@RequestMapping* is used on the class with the path argument, it defines the path for all methods implemented in that class. The *@RequestParam*, on line 11, is used to extract the request parameter. In line 12, the value of the *getUserProperty* method is retrieved from the ServerCommandProxy (scp). A UserID object is created using the value received as a parameter. Finally, the response object is wrapped in a ResponseEntity and returned with an HTTP 200 OK status code using *ResponseEntity.ok()*. This indicates that the request was successful, and the response body contains the user's secondary card number in JSON format. All other methods are implemented following this example. In addition to these methods, there are two cryptography methods. They perform encryption and decryption of a given set of data using the Security class.

Finally, one method returns a signed message. It is composed of the user identifier, the current time and the expiration time. These times are encoded with the Unix time timestamp. The user identifier is passed as a parameter. The current time is retrieved using *Instant.now().getEpochSecond()* which returns the current time in the correct format. The expiry time is the current time plus the validity time. It is set to twenty-four hours but can be changed to whatever is required. It is equal to the signed message validity time. These three values, in order and in hexadecimal format, are placed in a 32 byte array. Each of them is 8 bytes long. The remaining height bytes are padded to complete the array. The use of this signed message is explained and detailed in the chapter 5.

All this GET methods return values in a JSON format. It take the following form.

```
{
  "userList":["toml","chz","reto"]
}
```

Listing 4.3: Code - getUserList JSON response

```
{
  "userExists":true
}
```

Listing 4.4: Code - userExists JSON response

```
{
  "userID":"2108"
}
```

Listing 4.5: Code - getUserID JSON response

```
{
  "userBalance":52.45
}
```

Listing 4.6: Code - getUserBalance JSON response

```
{
  "encryptedData":"572df63be8b2bbc8b3535dc7a56c640d"
}
```

Listing 4.7: Code - getEncrypted JSON response

```
{
  "decryptedData":"112233445566778899aabbccddeeff00"
}
```

Listing 4.8: Code - getDecryptedData JSON response

```
{
  "signedMessage":"00000000000012340000000064d3ac48
                   0000000064d4fdc80000000000000000"
}
```

Listing 4.9: Code - getSignedMessage JSON response

5 | BLE communication

The Bluetooth Low Energy (BLE) is increasingly used for all kinds of applications today. It is an easy and reliable way to communicate between two devices. Modern smartphones are equipped with this technology and many devices work with the BLE. The given card reader (*Elatec TWN4 Slim*) supports LF/HF transponders and additionally BLE. As explained in the Analysis chapter, the availability of a Bluetooth module on this card reader influenced the choice of smartphone authentication technology. The Bluetooth Low Energy communication will be implemented in stages:

1. The phone connects to the card reader.
2. The phone passes the user identifier without authentication. (At this point, this procedure can be done with another application such as *nRF connect*)
3. The phone passes the user identifier without authentication but with Bluetooth encryption.
4. The phone passes the user identifier using a simple authentication protocol.
5. The phone passes the user identifier using an advanced authentication protocol.
6. The phone passes the user identifier using an advanced authentication protocol and without a network connection.

These stages allow the final objective to be achieved step by step. They make it possible to structure and guide the implementation.

In this chapter, in order to cover the whole subject of Bluetooth communication, an introduction to the BLE will be made to ensure understanding of the following sections. In addition, the authentication protocol provided by *Elatec* will be analysed to allow the specification of a protocol adapted to the needs of the project. Finally, the card reader and the mobile application will be presented in detail. All these parts will help to fully understand the implemented Bluetooth communication.

5.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a wireless communication technology designed for low power consumption. It operates in the license-free Industrial, Scientific, and Medical (ISM) band at 2.4 GHz. BLE divides this band into 40 channels, three of which are the primary advertising channels.

Bluetooth Low Energy devices can act as peripherals (sensors or beacons) or centrals (smartphones or computers). The peripheral device can advertise data to nearby devices to allow discover it. The central discovers these advertisement packets and possibly connects them to the peripherals. A device can also have connections to multiple devices.

Advertising is when a peripheral sends an advertisement packet to other devices. It sends these packets to the primary advertising channels. The central continuously scan these three channels looking for advertising data. These latter can allow connection or not, depending on the settings. Generally, the advertisement packet contains the device name, the transmission power (tx power), or even the services provided by the peripheral. It can send packets with specific advertising intervals going from 20 milliseconds to 10 seconds. The shorter the interval, the faster the device can be discovered, but the more power it consumes.

When a central has discovered a peripheral by scanning the advertising packet, it can send a connection request to connect. Once the response is received by the central, connection is established. BLE devices can exchange data using attributes (characteristics and services). It is the organization of the device data. Characteristics represent specific data values and services regroup one or more characteristics for easier management and discovery. Services generally regroup similar characteristics.

Different protocols manage the connection, the data transfers and the data. Bluethoot Low Energy has a generic profile called Generic Access Profile (GAP), which serves as a base profile for all BLE devices. It defines their basic requirements and enabling communication. GAP ensures security through cryptographic algorithms and the Security Manager. It also defines roles that BLE devices can operate.

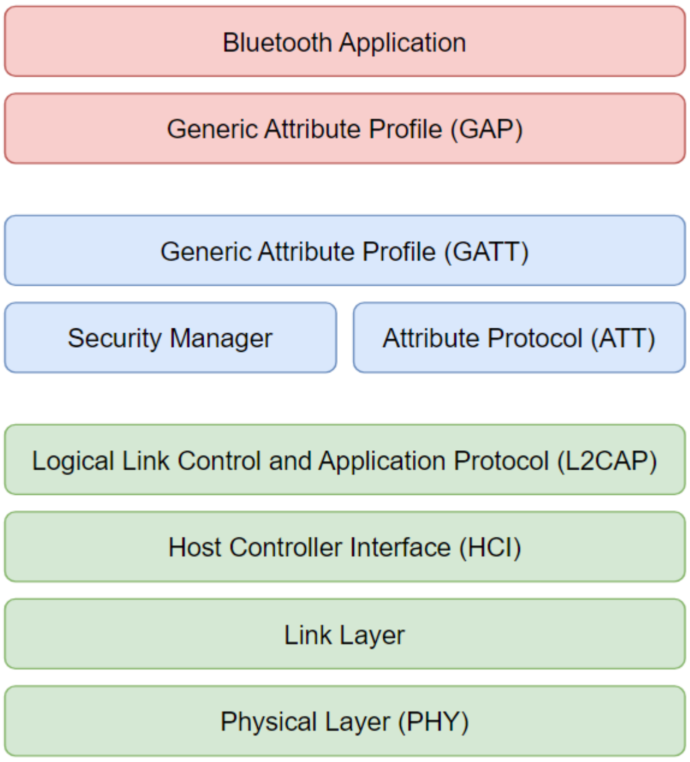

Bluetooth Application
Generic Attribute Profile (GAP)
Generic Attribute Profile (GATT)
Security Manager
Attribute Protocol (ATT)
Logical Link Control and Application Protocol (L2CAP)
Host Controller Interface (HCI)
Link Layer
Physical Layer (PHY)

Figure 5.1: BLE protocol layer (Source: Buildings Bluetooth Low Energy Systems [18])

Generic Attribute profile (GATT) is a high-level protocol built on top of the ATT. It defines a standard way for devices to discover, read, write, and exchange data between services and characteristics.

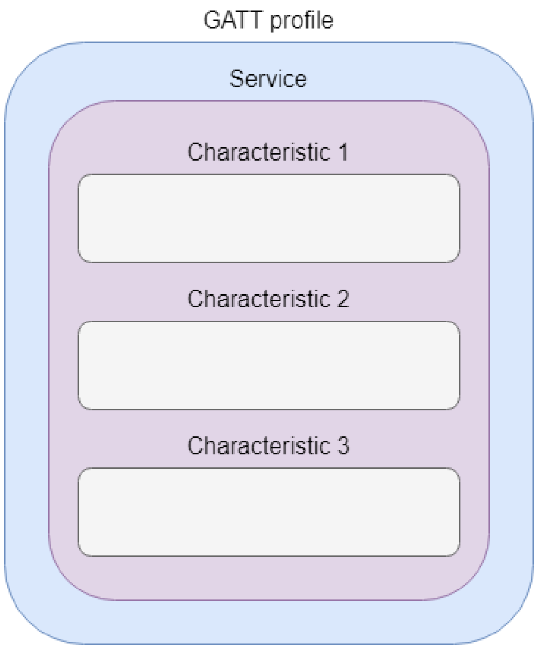


Figure 5.2: GATT profile model

Attribute protocole (ATT) is a protocol used to manage the exchange of data. It defines two roles: the server, which holds the data, and the client, which reads and modifies the data. A device can have both roles simultaneously. This protocol organises data into attributes, each of which represents a piece of information and has a unique identifier called handle on the server. It also has a UUID. It can be 16 bits long (a *SIG-adopted UUID* listed in the assigned number list [19]) or 128 bits long (a custom UUID). It also contains all the permissions for that attribute:

- **Read**: Permission to read characteristic values from the remote device.
- **Write**: Permission to write data to a feature on the remote device.
- **Notify**: Permission for the device to send notifications to the connected device when the value of a characteristic changes without confirmation.
- **Indicate**: Permission for the device to send indications to the connected device when the value of a characteristic changes, with acknowledgement required from the connected device.

Finally, BLE allows devices to change their communication range according to the requirements of the application, helping to save energy. The technology offers open and free access to its specifications, allowing communication to be implemented quickly [20].

5.2 Authentication protocol

In today's interconnected world, where information flows seamlessly across networks, ensuring the security and confidentiality of data is essential. Authentication protocols are critical in protecting sensitive information and providing reliable access control mechanisms. The protocol is designed to verify the identity of devices (smartphones and card readers). The protocol provided by the card reader supplier is analysed as a working basis.

5.2.1 Analysis

Elatec, the manufacturer of the given TWN4 card reader, proposes a solution for Bluetooth Low Energy communication [21]. They have developed their own application, *Mobile Badge BLE NFC*, which can be downloaded for free on Android or iOS. It simply sends the phone ID after authentication with the card reader. Some details of this process are given in the documentation, but the source code of the application or firmware is not accessible. To analyse this protocol, a Bluetooth sniffer (*Ellisys Vanguard*) is used. With this device it is possible to see all exchange Bluetooth packets in the air and it helps to understand the implemented communication protocol between the existing application and the card reader. For these tests *nRF Connect* will also be used. This application can scan and detect BLE devices. The behaviour of the firmware is thus tested without authentication. The card reader must be configured to support the authentication of the application. This can be done by following the procedure in the **BLE Credential Apps User Guide** from the *Elatec* development package [22]. The followings diagram shows the protocol given in the documentations.

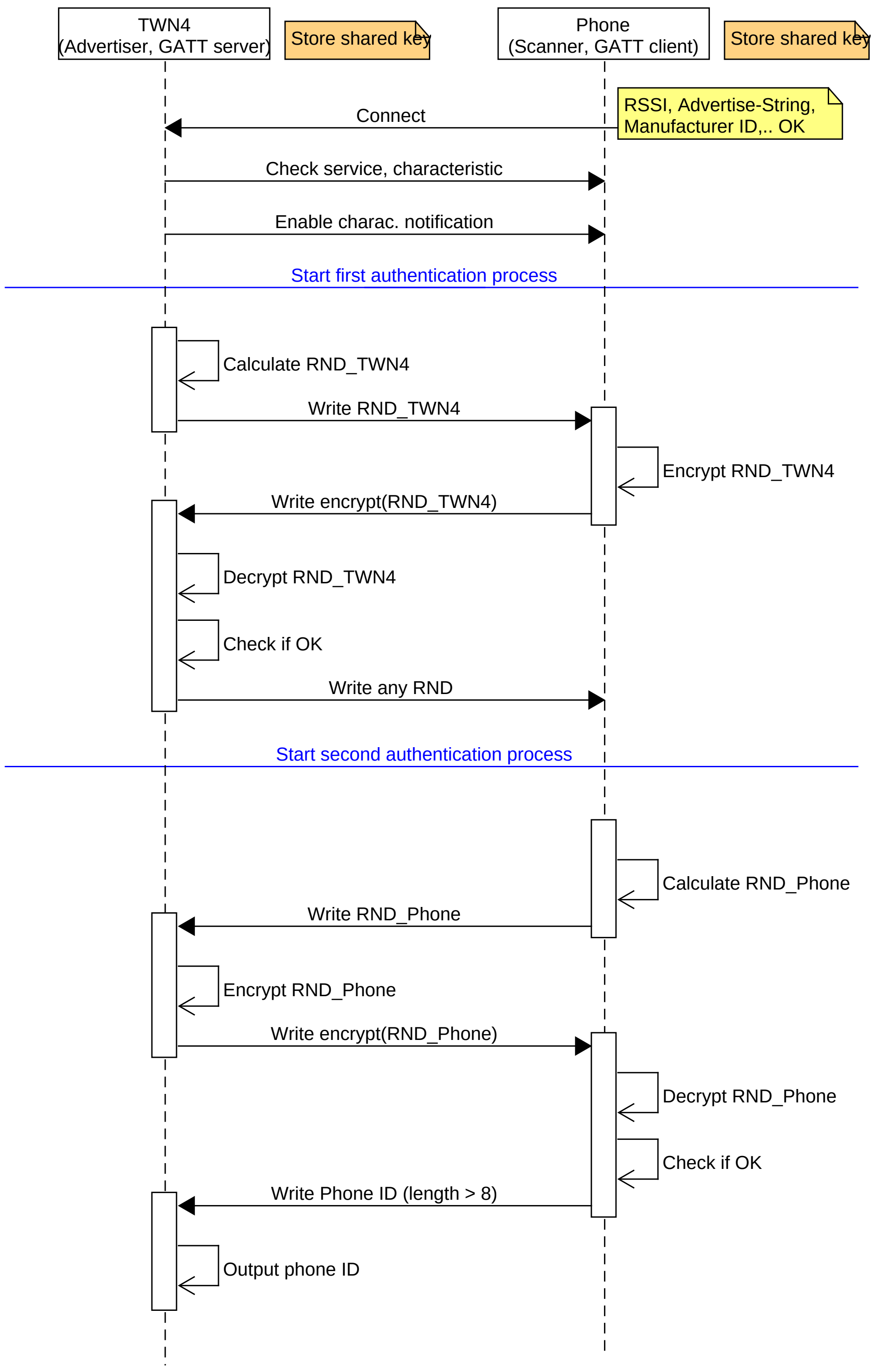
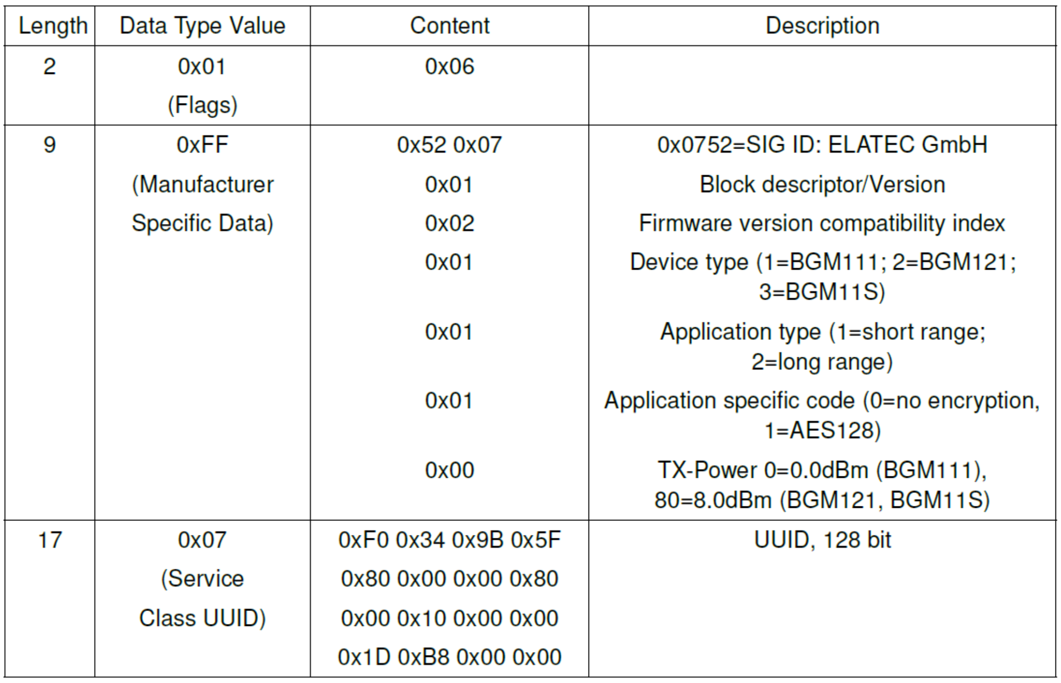


Figure 5.3: Elatec BLE authentication protocol *(Source: BLE Implementer's Guide @Elatec [23])*

The BLE/RFID module is connectable, which means that a nearby BLE device can establish a connection. It is also scannable. Another module can ask for additional data in a scan request.

If no device is connected to the card reader, the latter will advertise. It is the peripheral device. This mode allows other devices to see it and possibly connect to it (central device). To perform advertising, an advertising packet is sent on each of the primary advertising channels (channels 37, 38 and 39). These three packets (one for each channel) can be seen in the timing windows of the analyser. These packets are sent periodically by the device, in this case every 57 milliseconds.

In the documentation of the development package, the advertise packet is defined as follows



Length	Data Type Value	Content	Description
2	0x01 (Flags)	0x06	
9	0xFF (Manufacturer Specific Data)	0x52 0x07	0x0752=SIG ID: ELATEC GmbH
		0x01	Block descriptor/Version
		0x02	Firmware version compatibility index
		0x01	Device type (1=BGM111; 2=BGM121; 3=BGM11S)
		0x01	Application type (1=short range; 2=long range)
		0x01	Application specific code (0=no encryption, 1=AES128)
		0x00	TX-Power 0=0.0dBm (BGM111), 80=8.0dBm (BGM121, BGM11S)
17	0x07 (Service Class UUID)	0xF0 0x34 0x9B 0x5F 0x80 0x00 0x00 0x80 0x00 0x10 0x00 0x00 0x1D 0xB8 0x00 0x00	UUID, 128 bit

Figure 5.4: Elatec BLE advertise string (Source: BLE Implementer's Guide @Elatec [23])

In the *Ellisys* program, this information can be found in the raw data of the advertisement packet. It confirms that the *Elatec* application advertisement works as described in the documentation and in the Bluetooth definition.

When another device scans and enters the range of the advertiser, it will receive the advertising packet and read the content. It can be assumed that the application filters the discovered devices by their name. The application will try to connect only to a device with *ELATEC Gmbh (0x5207)* as manufacturer name. It also uses the advertiser's Received Signal Strength Indication (RSSI) to process the connection. *Elatec* proposes with its application a "RFID card" experience. The phone must be close to start the authentication process. It means that the RSSi must be greater than the value set.

Once these conditions are met, the connection can be established. Several LLCP packets are exchanged to achieve the connection. When the connection is established, the roles are defined in the detail windows. In the application they are the following

- **Application** : central and GATT server.
- **Card reader** : peripheral and GATT client.

Since the application is the GATT server, it provides its profile containing the custom service and its characteristics. The peripheral can now discover all the services provided by the central with a *ATT Find By Group Type Transaction* with a handle from one to the maximum value. Some standard services such as *Generic Attribute* or *Device Attribute* are found with their respective handle. Other *ATT Find By Group Type Transaction* requests are made with a different handle range to discover all available services. A particular service provided by the server is important, it has the UUID *0000B81D-0000-1000-8000-00805F9B34F1*. This private service is used to authenticate the phone.

This protocol uses double authentication with a shared 16-byte master key to perform the authentication. AES128 is used to encrypt the random 16 byte number. After the first exchange, if the decrypt number matches the send number, a new value is written to complete the first authentication. If the second exchange is correct, the phone writes the ID. After this sequence, the authentication is complete.

This process can be found in the exchange BLE packets. When the card reader writes a data, it uses a *Write Command Packet* on the private service. With this request, the client writes a new value on the choose handle and does not expect a response from the server. The server uses a *Notification Command* to notify an attribute change. It does not require an acknowledgement from the client. Working with no response is faster, but if a problem occurs, the data will be lost and the authentication process will fail. The smartphone ID is sent in clear text. The AES encryption is only used for authentication and not to protect the data.

The authentication protocol implemented by *Elatec* works as described in the documentation. When a other application (*nRF Connect*) tries to connect, the behaviour is not the same. The connection is established, but the card reader doesn't find the private service. So the client will not start the authentication protocol and it will disconnect the remote device.

A complete BLE exchange is available in the Git repository (*01_documentations/ble_sniffer*). It was captured using the Bluetooth Low Energy sniffer. Two cases were recorded for this analysis: an authentication using the *Elatec's Mobile Badge* application (successful) and an attempt using the *nRF connect* application (failed).

This protocol is a good basis for developing a authentication protocol for this project . Some modifications will have to be made to meet the requirements.

5.2.2 Specification

Based on the previous analysis, a protocol can be defined that meets the needs of this thesis. This protocol is based on the analysed one. It has been defined in discussion with the industrial partner to meet their needs in terms of security.

The protocol uses the same principle as the analysed one. Some modifications have been made. In this protocol, the GATT roles are reversed. The card reader is the server and the application is the client. This approach is more logical because the card reader already has a GATT profile defined. It does not need to be defined in the application.

The phone starts the procedure by sending a random number to be encrypted. The card reader is authenticated if the encrypted value returned is correct. In the *Elatec* solution, the TWN4 starts the procedure and the first device to be authenticated is the phone. The phone contains more sensitive data than the card reader. It has access to personal user data and communicates with the Print Manager server. The only sensitive data in the reader is the shared key for encryption, which is not accessible via BLE. It is more important to make sure that the smartphone is talking to the right device first. Then the process can continue and the phone can be authenticated.

On the next page, the complete sequence diagram of the protocol is shown. It contains all the components used for authentication.

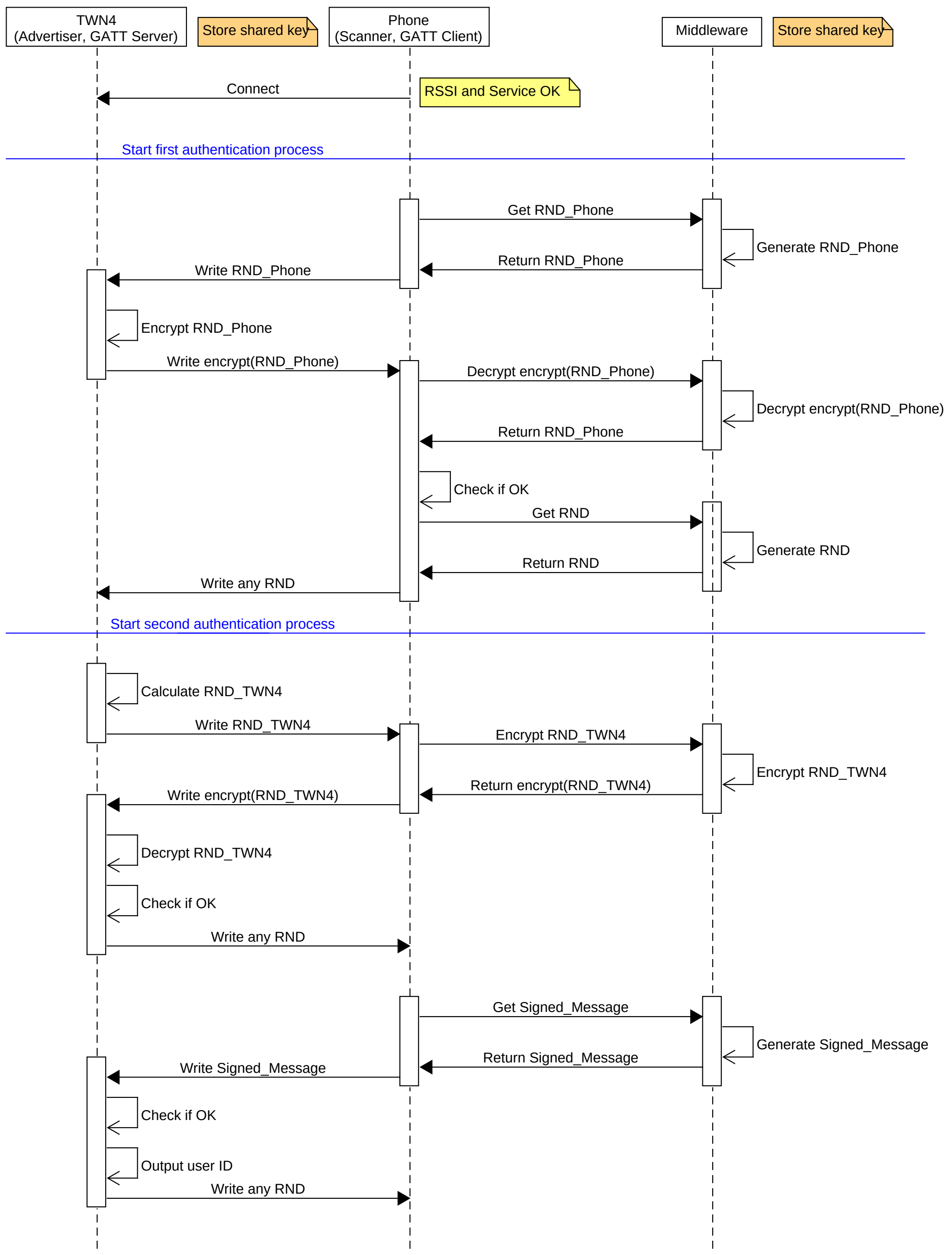


Figure 5.5: BLE authentication protocol

Once both devices have authenticated, the user identifier can be sent. A signed message is used in this protocol. It is generated by the middleware component in order to always have the least sensitive information in the application. The message consists of thirty-two bytes. The first eight are reserved for the user identifier. This length can be extended if longer identifiers are required. The next eight bytes are for the current time in Unix time. The expiration time is also in Unix format and is placed in the next eight bytes. These three values are hexadecimal. The remaining eight bytes are padded with zeros. In the future, the middleware could encrypt this message before sending it to the smartphone and next to the card reader to increase the level of security. The cryptography mode provided in the card reader and used in the web service has no padding. For this reason, the data length must be a multiple of 16. So, in anticipation of future implementation, 8 bytes are added at the end of the three valuable values.

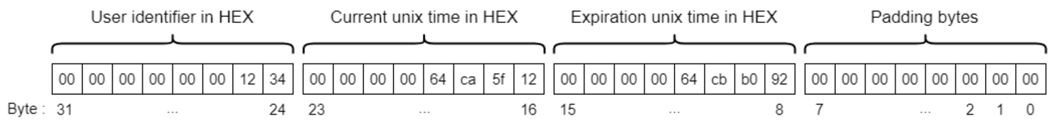


Figure 5.6: Signed message structure

The purpose of this signed message is to enhance the authentication process. Once the certified component generates a signed message, it can be stored in the mobile application to be reused until the expiration time. The current time makes it possible to update the time of the card reader. The latter can count the elapsed time using these system ticks. However, it counts the time from a given value. These updates allow the time and date of the reader to be adjusted.

With this protocol, a user cannot identify himself without a network connection. The encryption is initially managed by the web service. It will be able to store the signed message, but authentication can only be carried out with the possibility of encrypting or decrypting the reader's messages.

This protocol can always be evolved to increase the level of security, following advances in the field of security. As mentioned above, the signed message could be encrypted by the middleware component. This allows the reader to validate the signed message by decrypting it. In the solution analysed, the ID is simply sent in clear text. With a Bluetooth sniffer, this value can be retrieved and used for malicious actions. By encrypting the signed message, it is possible to hide this value from an eavesdropper. The BLE embedded encryption could be used for this.

5.3 Card reader

A card reader is a commonly used system for identification, payment or access control. Therefore, there are many choices on the market. For this thesis, the industrial partner provided a reader with which it works. This is the *TWN4 Slim* model produced by the company *Elatec*.

Figure 5.7: TWN4 card reader (Source: ©Elatec)

Firmware needs to be developed. It will be based on the documentation and samples provided by the vendor. It will be programmed in *C*, the language supported by the reader.

This section presents some useful features of this card reader and explains the specification and implementation of the firmware.

5.3.1 Features

The *TWN4 Slim* card reader supports all standard LF and HF technologies. It integrates a BLE module from *Silicon Laboratories* (BLE V4.2), which enables data communication and authentication [24]. It has three LEDs (red, green and blue) and a beeper to interact with the user. The TWN4 can read and write on its micro-USB port in keyboard mode. This is how it passes identifiers.

To use the reader correctly, the manufacturer provides a development pack (TWN4-DevPack451) [25]. It contains all the documentation, the applications needed to set up and use the reader and firmware examples. It also contains other useful files for running the TWN4. It can be downloaded from their website. Version 4.51 was used for this project. In this package, the **AppBlaster** application allows to configure the device with its interface. This tool can be used to set up the reader and load firmware. Its detailed operation is explained in the **AppBlaster User Guide**. [26]. This application will be used in this project to load the developed firmware.

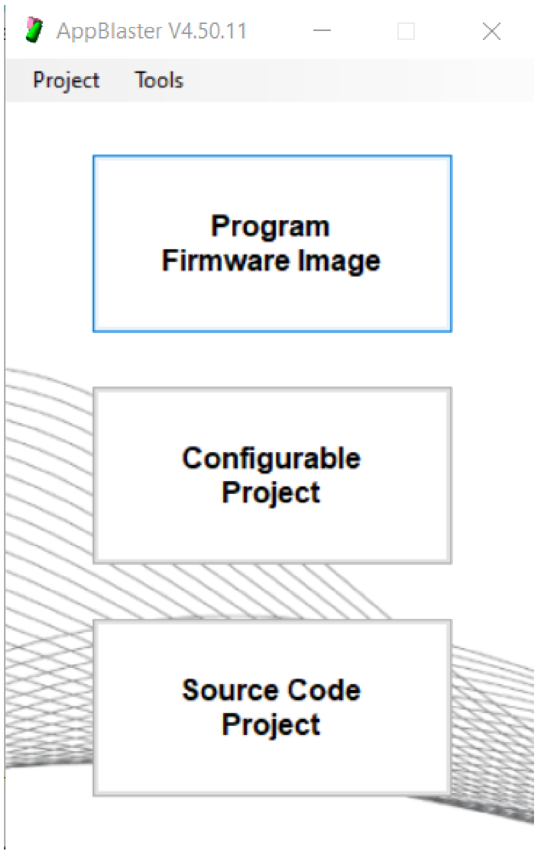


Figure 5.8: Elatec AppBlaster application (Source: ©Elatec)

A test application is also available. The **Director** allows to test all internal firmware functions and APIs of the reader. The **Director User Guide** describes all functions of the [27]. It is used to test some of the functions provided.

The behaviour of TWN4 can be controlled by the user with so called Apps. These apps are programs written in *C*. They have access to the *Elatec* API, which helps to develop RFID/BLE applications. These API functions are listed in the **API Reference** document [28]. Some basic ones and all those related to the RFID are provided. The functions to control the Bluetooth Low Energy module are available, such as BLEInit or BLECheckEvent.

Apps are not loaded directly into the reader. An image is created by the **AppBlaster** application. When it is generated, the firmware is compiled. The errors, warnings and notes are displayed. Unfortunately there is no debugger available. If a problem occurs in a API function, a message is written to the USB output (*APPSCREAM(()*. It can be printed on a document when the reader is connected to a computer. Debugging using only print text is commonly known as print debugging. It takes more time. The problem must be clearly understood so that markers can be added to specific places in the code. The program flow can be traced and the problem can be located. Changes can be made in the code, and processes can be restarted until the bug is fixed. Bluetooth has also been used as a debugger by writing values to a property. This value could then be read using the Bluetooth sniffer or an application such as *nRF connect*.

5.3.2 Implementation

In this section, the implementation of the card reader firmware will be presented. It includes card reading, BLE communication, the cryptography, time management and the authentication protocol. The appendix B lists all the tools and libraries used for this device.

Read card data

A sample App, available in the development kit, has been used as a base to start implementing the card reader firmware. The *C* file *App_STD204_Standard* allows the TWN4 to read data from all LF and HF tags. It uses the API functions provided. *SearchTag* searches for a transponder in the reading range of TWN4. It searches for all transponder types specified by the *SetTagTypes* function. If a transponder is found, the tag type, the length of the identifier and the identifier are returned. The data is converted and written to the USB port using the *HostWriteString* function.

As the card search function is in a while loop, the card will be found endlessly as long as it is in range of the reader. Logic has been added to avoid multiple card reads. When a card is found, a timer is started and the identifier is stored. When a new card is found, a comparison is made. If both identifiers are the same, the scanned card is ignored. The stored card identifier is set to zero when the timer expires. This data is stored in a buffer. If several different cards are read simultaneously, the previous value is not overwritten and all identifiers read are printed.

Modifications have been made to adapt this solution to the needs of the project. All variables have been made global. Logic that was in the while loop has been placed in the *scanCard* function to make the code more organised.

A problem encountered was the slowness of the firmware when other functions were used along with card reading. There are several reasons for this slowdown. The beeper and LED functions are blocking functions. Avoiding them changing states too regularly can improve the speed of execution. A simple beep of a few hundred milliseconds or a colour change is tolerable, but more than that becomes problematic. Another reason is that at each iteration of the while loop, all tag types are searched. If the reader only reads a RFID card this is not a problem, but with more complex firmware the program will slow down.

Tests have been made with simple Bluetooth Low Energy identification without security in addition to card reading functionality (firmware version V1.0-without_security). The time for the BLE identification (connect, read identifier and write to USB output) was calculated with the different modifications. The following results were obtained:

Firmware cases	Average time [s]
Without modifications	13
Without led and beeper blink	12.5
Without all kinds of tags	3
Without all kinds of tags and led and beeper blink	2.5

Table 5.1: Average time for different firmware cases.

To optimise the programme, only the MIFARE tags have been retained. The industrial partner only uses this technology with this card reader. Also, only a beep and a colour change of the LED have been implemented. With these changes, the time for a simple Bluetooth identification is about the same as for a card.

BLE communication

The Bluetooth Low Energy (BLE) module is used with the functions provided by the API. The implementation of the Bluetooth communication was done step by step. Without a debugger, it was easier to validate each small step.

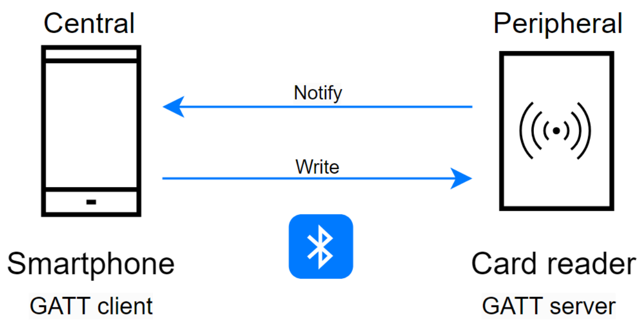


Figure 5.9: Diagram BLE communication

The smartphone will act as the central point in this communication. It will initiate the connection request with the TWN4. The latter is therefore the peripheral. It will accept incoming connection requests after advertising.

The first step was to advertise by calling the *BLEInit* function with the parameter BLE_MODE_ADVERTISEMENT. This allows advertising without encryption or bonding. The default advertisement string is used. It contains the manufacturer name, the device name and especially the UUID of the service provided. This format is suitable for the thesis, but it can still be customised using the *BLEPresetUserData* function. This function sets the user data string for advertising. All readers have the same device name **TWN4 BLE**. If a unique name is wanted for each reader, the unique identifier of the reader can be used. This can be obtained using the API function *GetDeviceUID*. With an application such as *nRF connect*, it is now possible to connect to the reader. Its services and these characteristics are visible and can be modified according to the respective permissions. All available permissions are described below.

UUID name	Type	Permissions
UID1	Service	-
UID2	Characteristic	Read, write no resp., notify, indic.
UID3	Characteristic	Write no resp., notify, indicate
UID4	Characteristic	Read, notify, indicate
UID5	Characteristic	Read, write, notify, indicate

Table 5.2: Custom GATT Services defined within Firmware (Source: ©Elatec)

The *BLEInit* function provides various initialisation parameters. It is possible to create a custom configuration for the BLE module using the *BLEPresetConfig* function together with a TBLEConfig object. The following configuration has been created for this project:

```
// BLE parameters
TBLEConfig BLEConfig =  {
 .ConnectTimeout = 12000,    // Timeout of an established connection in ms
 .Power = 20,                // TX power: 0 to 80 (0.0dBm to 8.0dBm)
 .BondableMode = 0x00,       // Bonding: 0 = off, 1 = on
 .AdvInterval = 200,         // Advertisement interval: 20ms to 10240ms
 .ChannelMap = 0x07,         // Advertisement Bluetooth CH37 + CH38 + CH39
 .DiscoverMode = 0x02,       // LE_GAP_GENERAL_DISCOVERABLE
 .ConnectMode = 0x02,        // LE_GAP_CONNECTABLE_SCANNABLE
 .SecurityFlags = 0x00,      // Security requirement bitmask
 .Passkey = 0x00000000,      // Passkey if security is configured
}
```

Listing 5.1: Code - BLE custom configuration

The transmission power doesn't need to be high because the user needs to be close to the printer to authenticate, as with a card. It also increases the security of the system. Because the smartphone must be closed to the reader to connect, the risk of replay, relay or Man in the Middle attacks is reduced. These would require the attacker to get close to the phone or to transmit with high power.

Power consumption is not an issue as the card reader is not battery powered. A low advertising interval can therefore be chosen. Any device can discover the reader (LE_GAP_GENERAL_DISCOVERABLE). It can be scanned and connected (LE_GAP_CONNECTABLE_SCANNABLE). The security features are not used.

In the second step of the implementation, the developed mobile application connects in BLE and transmits an identifier which is output (firmware version V1.0-without_security or firmware version V1.1-without_security). The *BLECheckEvent* function returns the current event of the BLE module. This allows to know the status of the Bluetooth module. Three events have been used to implement this step BLE_EVENT_. CONNECTION _OPEN, BLE_EVENT_CONNECTION_CLOSED and BLE_ EVENT_GATT_SERVER_ATTRIBUTE_VALUE. To aid debugging, specific beeps and LED colour changes have been added to indicate when a device is connecting or disconnecting. The reader emits short beeps when a connection is made and the led starts flashing red. When the device is disconnected, the LED stops flashing and returns to green. A long beep is also emitted. As explained above, this feature has been removed in future implementations to reduce processing time.

Once the connection is established, the application writes the user identifier into the *UID5* property (UUID: *495f449c-fc60-4048-b53e-bdb3046d4495*). The BLE_EVENT _GATT_SERVER_ATTRIBUTE_VALUE event is fired. The *BLEGetGatt- ServerAttributeValue* function returns the value in a byte array. This value can be hardcoded. However, the *BLEGetGattServerCharacteristicStatus* function returns the handle of the last modified value. This ensures that the correct value is always retrieved when multiple characteristics are used. If this function is called after the BLE_EVENT_GATT_SER VER_ATTRIBUTE_VALUE, bit 15 of the handle attribute must be set to 1. Once the value is read, it can be written via USB using the *HostWriteByte* function.

The mobile application coded in JavaScript has no type for the variables. The user identifier, encoded in hexadecimal, is stored as a string value. The hexadecimal format is not respected when writing to the card reader characteristic. The value read, placed in a byte array, takes the form of *0x1, 0x2, 0x3, 0x4, ...* instead of *0x12, 0x34, ...*. It works to write this value in keyboard mode. For the next operations required for authentication and encryption, the value needs to be transformed. To do this, the byte array must be transformed by merging every two bytes into one byte using the transformByteArray function.

To exchange data with the card reader, the application always uses the *UID5* characteristic, because it is the only one that has granted write permission. The GATT server provided cannot be customized to remove unused properties or modify permissions. The card reader notifies the smartphone of the change to the characteristic in order to transmit data. The value is modified by calling the *BLESetGattServerAttributeValue* function with the corresponding handle attribute. Setting bit 15 of the handle to 1 sends a notification to the connected device. This is a two way communication.

The simplest iteration of the card reader firmware is functional. On the git repository, this version is tagged *V1.0-without_security* or *V1.1-without_security* (only the mobile application changes between these two versions). The code is not cleaned or commented in these versions as it was not the final version. This firmware makes it possible to connect to a device, read a modified characteristic and print its value. This operation can be done with the first iteration of the mobile application and other applications (e.g. *nRF connect*). This is possible because there is no security to implement. The identity of the connected device is not verified. This solution provides identification but not authentication. This firmware could be used as a final solution, but it does not protect against any kind of To fulfil objective three, the authentication protocol previously specified must be implemented. The Bluetooth communication will not be modified for this. It will be used to transfer data between the two connected devices.

Cryptography

For the authentication process to be successful, the messages exchanged must be encrypted or decrypted. The card reader provides a cryptography API. It can do Triple-Data Encryption Standard (DES) or AES. While both aim to protect data, they are not identical in design or security. A development of the more established Data Encryption Standard, Triple-DES uses DES three times in quick succession with different keys to increase security. However, it is less preferred in modern applications due to its slow performance and vulnerability to certain attacks. AES uses fixed size blocks, making it a highly effective and secure encryption standard. AES has replaced triple-DES as the encryption method of choice across a wide range of industries due to its higher security and speed. The AES was therefore chosen with a key length of 128 bits (the only key length available in the card reader).

The cryptography module uses the Ciphered Block Chaining (CBC) method. In CBC mode, each encryption operation depends on the previous one. This is achieved by including the Init Vector (IV). The first CBC operation usually works with a Init Vector set to zero. For encryption, a plaintext block P is logically XORed with this Init Vector before it is encrypted. The result is an encrypted block C, which serves as IV for the next operation. This method increases the level of security. The Init Vector is modified at each step. As it is not always the same device that is connected to the reader, the connected devices have no way of tracking the IV value and the encryption can be wrong. So the IV must be reset after each process.

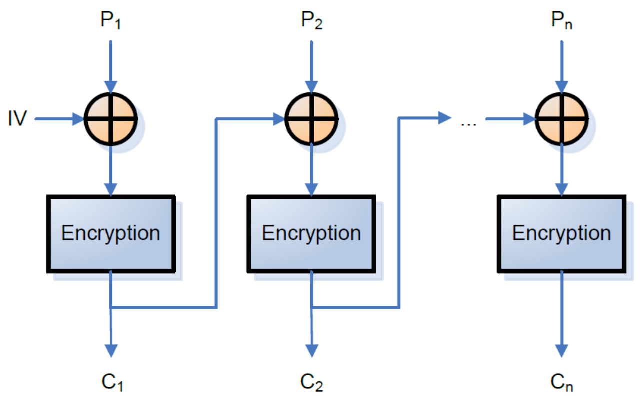


Figure 5.10: CBC enciphering scheme (*Source: ©Elatec*)

Before use, the cryptography API must be initialised using the *Crypto_Init* function. The key is passed to the cryptography method and assigned to an environment after initialisation. The functions for encryption and decryption are configured for the specified cryptography mode after initialisation. The same environment is always used for this work.

The *Encrypt* and *Decrypt* methods can now be used to implement the authentication protocol. These functions have no padding. The length of the given byte array must be a multiple of 16 to work. This is also the minimum size to work.

For the authentication protocol, the key is generated once and hard-coded in both the card reader and the Middleware. The TWN4 is equipped with a security module to store these keys. It offers several features:

- Solution providers send firmware images and project files containing secret data even over public communication channels, e.g. email.
- Solution providers ensure, that their TWN4 in the field are programmed with wanted firmware images only.
- TWN4 cannot be programmed with firmware, which is intended for use in a different application or by 3rd party.
- Protect TWN4 against being configured by others.

The best practice would be to store these keys on this module, but the reader's Bootloader version must be above version 1.20. The card reader used for this project is only version 1.08 and this Bootloader cannot be upgraded. It is included with the purchase.

The key for this thesis is hardcoded into the *C* file. As an image is created and loaded into the reader, the key value is not directly visible, but the risks are not non-existent and security is not fully guaranteed.

For future use, it is recommended to use card readers with a Bootloader version higher than 1.20 and to register the keys according to the procedure provided by the manufacturer [29].

Time management

In the defined authentication protocol, the expiration time must be controlled to invalidate the old signed message. The reader cannot provide the time and date. However, it can count the system ticks since the start of the firmware. The *GetSysTicks* function returns the specified number of system ticks in multiples of 1 millisecond. It is possible to calculate the date and time from a given date.

The Unix timestamp is used for ease of transfer and comparison. It corresponds to the number of seconds elapsed since 1 January 1970. The *readerCurrentTime* is stored as a 64-bit value. With a 32 bit value, the *Year 2038 problem* will appear. The integer value will overflow on 19 January 2038 at 03:14:07 UTC, setting the value to $-(2^{31})$. This value corresponds to 13 December 1901, 20:45:52 UTC.

The function *updateTime* is called in the while loop of the main program. It counts the ticks elapsed since the last call and stores them in the variable *elapsedTicks*. The current time is incremented if this value is greater than 1000 (1 second). The *GetSysTicks* functions return an unsigned long value (32 bits). Its maximum value is (2^{32}) milliseconds (about 1193 hours). The returned value will be reset to 0 after this number of system ticks. This can be managed by storing the number of the last tick and comparing it with the new value. The difference is added to the elapsed ticks value if the new value is greater than or equal to the previous value. Otherwise the elapsed time must be calculated using the following formula:

$$elapsedTicks = elapsedTicks + ((2^{32} - lastSysTicks) + sysTicks)$$

The ticks passed before the restart at zero and the return system ticks (the ticks passed after the restart at zero) are added to the elapsed ticks.

Time management is implemented. The reader's time is updated with the current time of the signed message if the latter is valid. The time and date will not be accurate, but the more users use the reader, the more accurate the time will become. One problem always remains. The current value is lost when the card reader is switched off. When it is restarted, the time will be the original encoded time. The card reader has a 48kByte flash memory. The current time could be stored in this memory. In the past, this solution allowed for less. However, it is not possible to know the time without power. It also has to wait for the first valid message to update the time. This solution has not been implemented as it is of minimal benefit and may slow down the operation of the firmware.

Authentication protocol

The authentication protocol involves the exchange of data to authenticate the reader and the smartphone. It was decided to use a State Machine (SM) to track and control the authentication sequence. This allows a simple system to handle complex processes. It is a computational model used to describe the behaviour of a system or program. It consists of a finite set of states, transitions between those states based on events or inputs, and actions associated with each transition. The diagram is visible on the next figure.

The authentication protocol is implemented. The different states of the SM have been defined. Before coding the machine, the actions performed in each state should be described verbatim. The transitions were also defined. This work facilitates the implementation of the state machine. The following diagram was created from this analysis. It contains all the states with a short description of their functions, as well as the transitions.

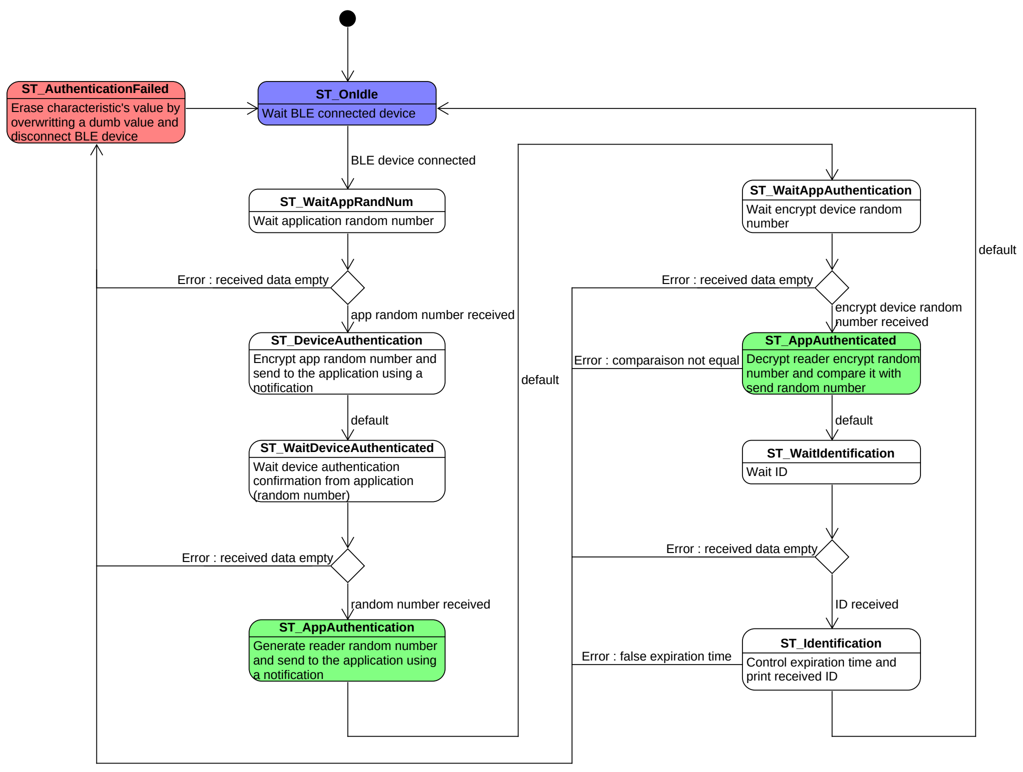

ST_AuthenticationFailed
Erase characteristic's value by overwritting a dumb value and disconnect BLE device
ST_OnIdle
Wait BLE connected device
BLE device connected
ST_WaitAppRandNum
Wait application random number
Error : received data empty
app random number received
ST_DeviceAuthentication
Encrypt app random number and send to the application using a notification
default
ST_WaitDeviceAuthenticated
Wait device authentication confirmation from application (random number)
Error : received data empty
random number received
ST_AppAuthentication
Generate reader random number and send to the application using a notification
default
ST_WaitAppAuthentication
Wait encrypt device random number
Error : received data empty
encrypt device random number received
ST_AppAuthenticated
Decrypt reader encrypt random number and compare it with send random number
Error : comparaison not equal
default
ST_WaitIdentification
Wait ID
Error : received data empty
ID received
ST_Identification
Control expiration time and print received ID
Error : false expiration time
default

Figure 5.11: Reader's SM diagram

A states enum defines all states. At initialisation, the *currentState* variable of type *States* sets the state to ST_OnIdle. It remains in this state until a BLE connection is made. When a device is connected, the authentication procedure starts with the reader authentication. The TWN4 waits for a random number from the application in the ST_WaitAppRandNum state. The latter has no logic other than to set the state to ST_DeviceAuthentication when a value is written to the *UID5* property. The received random 16-byte number must be converted to the correct format. It can then be encrypted and sent back to the phone with a notification. It then waits for the application verification response in the ST_WaitDeviceAuthenticated state. Reader authentication is complete when the phone sends a new random number.

Application authentication can be started (ST_AppAuthentication) by sending a generated random 16-byte number to the application. This number is generated using the *stdlib* rand function. A seed is first set using the number of elapsed ticks. It can generate different sequences of random numbers each time, as long as the system ticks are different. This makes it more difficult to predict the random number using eavesdropping messages. Once the seed is set, the 16 bytes are generated randomly. The %256 ensures that the random value is in a byte range (0 to 255). Once the number has been sent, the SM forwards the application's response in ST_WaitAppAuthentication state.

The smartphone returns the encrypted random number. This puts the device in the ST_AppAuthenticated state. The received data is decrypted after the format is transformed. If the decrypted value does not match the sent value, it goes to ST_AuthenticationFailed and the process is stopped. Otherwise, a random number is sent to confirm the authentication. Both the card reader and the application are now authenticated. The device enters the ST_WaitIdentification state to wait for the signed message. The boolean *receivedDataLength64* is set to true. Up to this state the received value was 16 byte values (32 bytes in the wrong format). The signed message is 32 bytes long and takes 64 bytes before format conversion.

When the signed message is received, the state machine enters the ST_Identification state. In this state the received data is parsed. The user identification is copied into an array of 16 bytes. After transforming the data into the correct format, the current and expiration times are copied into array bytes. This value is converted to uint64_t and compared to the current reader time. The expiration time must be greater than or equal to the current time of the message and the current time of the reader for the time to be updated. If not, the procedure is aborted and the state ST_AuthenticationFailed is set.

The user identifier can be written to the USB. The authentication protocol is successfully completed. Before returning to the ST_OnIdle state, the card reader notifies the application with a random number to indicate the success of the procedure.

Several functions manage the change of state and the logic. The first (*chooseSM-stateAttributeChanged*) is only called when a message is expected. The inner switch contains all waiting states. If a message is received in the checkBLEEvent function and the value is read correctly, the machine can go to the next state. Otherwise it goes to the ST_AuthenticationFailed state and the procedure is aborted. The second function (*chooseSMstate*) contains all the other states and their logic is explained above.

Two events can abort an authentication procedure. If a comparison or attribute value read fails as explained, or if the timeout is exceeded. A ten second timer is started when a device connects. When the timer expires, the ST_AuthenticationFailed state is automatically set. This prevents unwanted devices from remaining connected to the peripheral indefinitely. In addition, if an unauthorised device connects and fails to complete authentication, it is automatically disconnected when the timer expires.

If authentication fails for any reason, the card reader writes a dummy value in the attribute to overwrite the data in the feature. It avoids leaving a signed message or encrypted data in the attribute. If the operation is successful, the last value written is a random number that does not need to be overwritten.

The device is then disconnected. The *deviceDisconnected* function must be called. Usually it is called by the BLE_EVENT_CONNECTION_CLOSED event when the central disconnects. Sometimes the BLE module did not work correctly after an authentication failed due to the sudden disconnection. To fix this, the module is reinitialised after a failure. The state machine finally returns to the idle state and the card reader is ready for a new authentication.

5.4 Mobile application

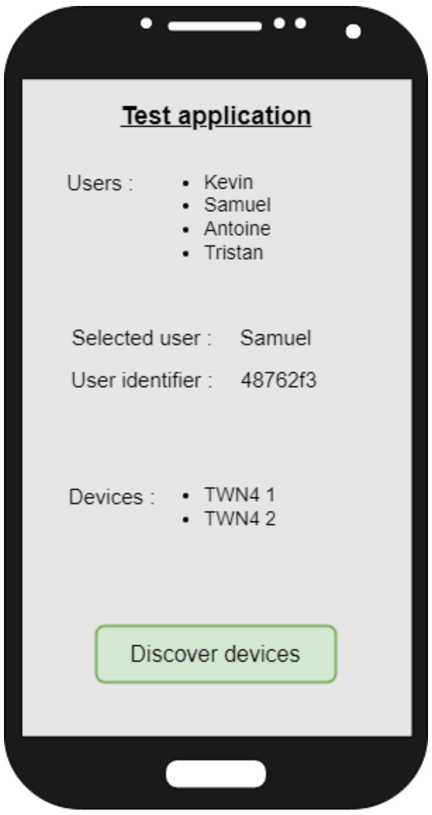
A simple mobile application has to be developed to test the communication. It replaces the *Polyright's* application. The technology must be chosen for this component. The design is unrestricted. If this Proof of Concept is used in the future, only the logic will be integrated into the existing environment.

This section contains the specification of the application. The choice of technology is also explained. Finally, the implementation is presented. The appendix B lists all the tools and libraries used for this device.

5.4.1 Specification

This application is a test application. It must allow us to develop and test the PoC. To do this, it was decided to be able to authenticate with different users on the same application. This makes it possible to simulate several users with a single smartphone.

The test application must allow the selection of the desired user. To do this, it must retrieve the list of users from the Print Manager server. The user identifier must also be retrieved.



Test application

Users :
- Kevin
- Samuel
- Antoine
- Tristan

Selected user : Samuel

User identifier : 48762f3

Devices :
- TWN4 1
- TWN4 2

Figure 5.12: Mobile application design

The application communicates with the card reader via BLE. Its main purpose is to connect to the card reader and authenticate itself. Once authenticated, it can transmit the chosen user identifier. To facilitate development and debugging of Bluetooth communication, all detected card readers are displayed in a list. It will be possible to connect and authenticate to the desired reader by bringing the smartphone close to it. The purpose of this feature is to allow the user to see no difference between using his card or his smartphone. This also allows the user to select the desired printer when there are several nearby.

The image above shows a design for the application. It is not the final version, but it should help to guide the implementation of the user interface. This design is inspired by the specification and will evolve with the components provided by the application framework used.

5.4.2 Choice of technology

The choice of technology is free. For the Middleware component, there are two competing options: *Android Studio*, used by the school, and *React Native*, used by the industrial partner. *Android Studio* is an integrated development environment specifically designed for the creation of native Android applications in Java or Kotlin. Android and iOS applications require different codebases to be created. React Native provides a unified way to build cross-platform mobile apps using React and JavaScript. While streamlining development, this single codebase can provide native-like components for iOS and Android. However, the JavaScript bridge can result in a small performance hit. It also provides pre-built components.

Since *Polyright* could take the logic of this application and integrate it into their own, it was decided to use *React Native* to develop this test application. In addition, performance is not required for this application. Therefore, it does not need to be optimised, which would have favoured *Android Studio*.

React Native uses React and JavaScript to build mobile applications, maintaining a bridge between the JavaScript code and the native iOS and Android components. The JavaScript script defines the structure and functionality of the app, which effectively handles User Interface (UI) updates using a virtual DOM. The pre-built components in React Native correspond to native UI elements. The bridge facilitates communication between JavaScript and native modules, allowing interaction with device functionality.

This application needs to use Bluetooth Low Energy to communicate with the card reader. There are several libraries available online for using Bluetooth. The two main ones are **react-native-ble-plx** and **react-native-ble-manager**. They offer the same main features:

- Observing the adapter state of the device's Bluetooth
- Connecting to peripherals
- Scanning BLE devices
- discovering services
- Discovering, reading and writing characteristics
- Observing indications and notifications
- Negotiating MTU
- Reading RSSI

There are some minor differences in certain domains. Since there are no major differences that would favour one over the other, it was arbitrarily decided to use the **react-native-ble-plx** for this thesis [30]. It is also a more widely used library. It is not up to date for the latest version of React Native. Two warnings appear during initialisation. They do not affect the operation of the application. The solution to remove them has not been found. They are ignored by the application. They still exist, but are no longer displayed on the user interface.

5.4.3 Implementation

This section presents the implementation of the mobile application. It includes all these components, its custom hook and the authentication protocol. All the tools and libraries used to develop this part can be found in the appendix B.

To set up the project, the **React Native CLI Quickstart** procedure was followed [31]. The development OS and target OS can be selected. Once the project is created, all the required dependencies can be downloaded using Node.js.

- **axios**: HTTP client for Node.js.
- **buffer**: Handle binary data and perform low-level manipulations.
- **react-native-base64**: Base64 encoding and decoding tools.
- **react-native-ble-plx**: BLE stack and APIs.
- **react-native-permissions**: Permissions API on iOS, Android and Windows.

If the latest version of Java Development Kit is used, it need to change the *Gradle* version of the project so it can recognize the JDK. This can be done by going to */android/gradle/wrapper/gradle-wrapper.properties* and changing the distribution value to update the *Gradle* version. The latest version of *Gradle* can be used.

Expo can also be used to start a React Native project. It is a framework and platform for building and deploying applications using JavaScript and React. It simplifies development with tools and workflow. A client is provided for testing the application. *Expo Go* allows to open applications served by *Expo CLI* and to execute projects faster. It is an application installed on the development device. A development server can be run. The application can scan a QR code to connect to the server. Once connected, the developed application is automatically updated and can be tested.

This mobile application was initially launched using *Expo*. Many problems were encountered during installation. The BLE is now supported by *Expo*. Previously it had to be excluded when using Bluetooth Low Energy. So there is very little documentation and experience. Unable to get a simple device discovery application to work, the **React Native CLI Quickstart** was eventually preferred to *Expo* for the rest of the work. The *Expo* test can be found on GitHub between the commits *753da72 - Initial application created* and *5467d6c - Initial React Native CLI project created*.

The project is now set up and implementation can begin.

App component

The *App.tsx* file defines the main application component. It contains the user interface of the NetPrinting identification application. This JSX component renders a view containing the title, custom user, BLE component and credit text. The JSX is a syntax extension for JavaScript commonly used in React Native. It allows HTML-like code to be written inside JavaScript files. It makes it easier to define and render the user interface. This component therefore contains the UI.

All components are combined into a single user interface element. This is wrapped in a view component, analogous to a *<div>* element in HTML. It acts as a container and therefore has the container style. In React Native, the flex property controls component spacing. If two components have the same value, they share the space, while higher values mean more space. In UI, the container takes all available space. A stylesheet defines all the styles used in this file.

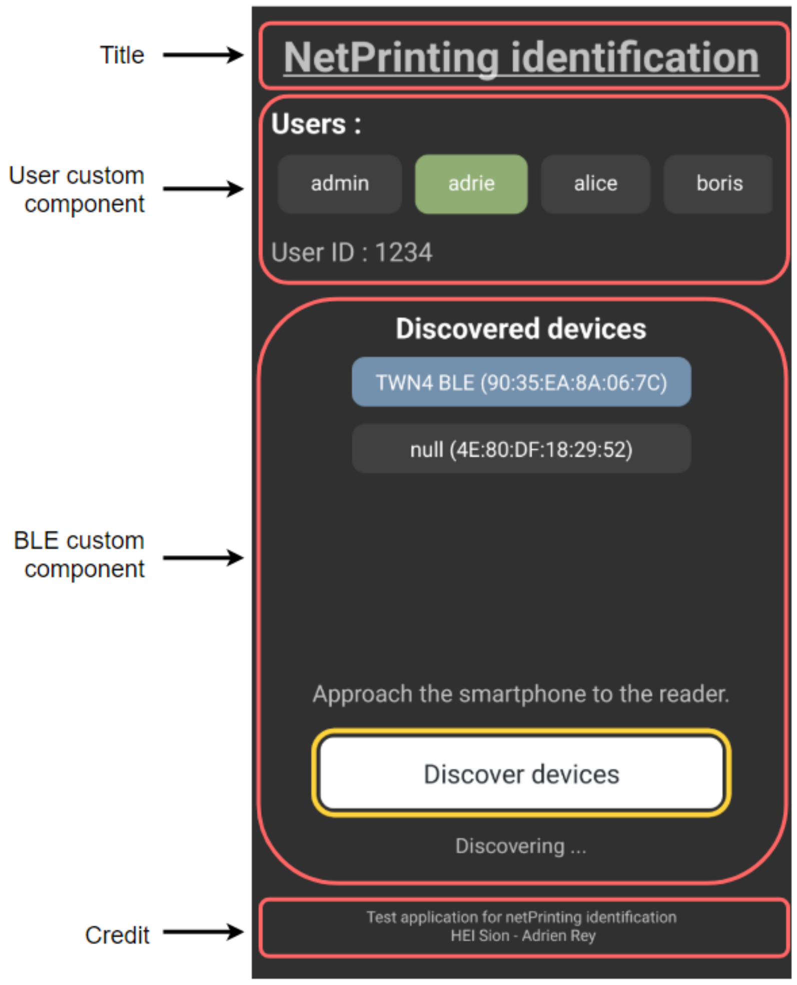


Figure 5.13: Mobile application final design with components

The figure above shows the final design of the application with all the components in the user interface. The red rectangles show the different components. Both custom components will be described in the following subsection.

User custom hook

The useUser file creates a custom React hook called useUser. Hooks in React Native allow function components to access state and other React features without writing class components. This makes it easier to reuse stateful logic between components. There are built-in hooks:

- **useState**: Manages state within a functional component and permits data storing and updating.
- **useEffect**: After component rendering, takes care of side effects like data fetching.
- **useContext**: Access context values.
- **useReducer**: Alternative to useState, manages more complex state logic using a reducer function.
- **useMemo**: Memorizes the result of a function.
- **useCallback**: Memorizes callback functions.
- **useRef**: Creates a mutable reference that persists across renders.

The most commonly used hook is the state. It will be used several times in this application to store data. Below a useState hook is initialised. It contains the value (*userName*) and its setter (*setUserName*). The parameter passed in the state is the initial value. Once initialised, the state can be used and modified in the component. A major advantage of using a hook is that the state is automatically updated when it is displayed in a component.

```
const [userName, setUserName] = useState('');
```

Listing 5.2: Code - Initialization of a useState hook

It is also possible to create a custom hook as useUser. Custom hooks always start with *use*. It is used when multiple components need to use logic and one of the built-in hooks does not fit. This custom hook manages states related to user information, including usernames, user IDs and a list of users. It uses React's useState and useEffect hooks to manage and update these states based on conditions.

The useUser hook defines the *onChangedList* function. It is used to change the selected user (*userName*). This function is returned with its state and the other two states: *userID* and *userList*. The effect hook is designed to retrieve and update user related information based on changes to the userName state variable. The function is called at the beginning and every time this state changes. If no user is selected, the user list is retrieved from the Middleware component using the *axios* GET method and stored in the appropriate state. If a user is selected, its identifier is retrieved from the Middleware component with a REST GET call. The state is updated and the value is passed to the BLE component using the *setUserID* functions provided by the latter.

User component

As seen in the app component section, it returns the user list. This list is retrieved using the user's custom hook. The initialization below allows to use the hook. The three states can be used as variables.

```
const [userName, onChangeUserName, userID, users] = useUser();
```

Listing 5.3: Code - Initialization of the useUser hook

This component displays the user list using a flat list. It creates a horizontally scrollable list. Each user is represented by an item defined by the *renderItem* function, which contains the user name as displayed text. This function creates an item with a touchable opacity wrapper, making it a button. When a user's button is pressed, it is selected and the style changes to highlight that user. Their identifier is also selected and printed below the flat list. The highlighted item and the user identifier are updated when a new user is assigned. This user component is rendered in the main application component.

BLE component

This last component is the most important. It makes Bluetooth communication possible. It also allows authentication according to the protocol.

A button is rendered in this component. It starts the discovery process. To do this, a user must be selected. First, permissions are requested using the *requestPermission* function. Access, BLE scanning and BLE connect, are requested to be able to use it. The request permission function is called each time the button is pressed. However, if the permissions have already been granted, the user will not be prompted again. Once the permissions are granted, scanning can be started by calling *scanFordevices*. The **react-native-ble-plx** library provides a BLE manager. It contains all the methods you need to interact with a device. Using the *startDeviceScan* function from the manager, it scans all devices, but only a specific one that is added to the discovered devices list (useState hook). To be added to this list, a device must provide the custom service *5a44c004-4112-4274-880e-cd9b3daedf8e*. The application is only interested in the TWN4 card readers. If it is not already in the list, the discover device will be added. When the list is filled with new devices, it is sorted in descending order by the device's RSSI.

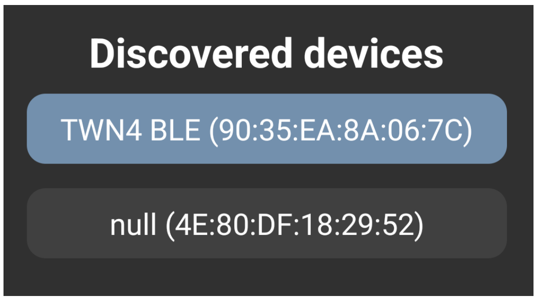


Figure 5.14: Application discovered device list

This list is displayed in a flat list in the BLE component (figure above). The device name is printed together with the MAC address. By default, all TWN4 have the same device name. The address is unique and allows us to differentiate between them. There are no pressable elements in the user list and they are displayed vertically. The devices are sorted, with the closer one at the top (with the larger RSSI). The latter is highlighted with a special style. The smartphone continuously discovers devices. Discovered devices can be discovered more than once. The list is then updated with the data of the new device and sorted.

To connect to a card reader, the smartphone must be approached by it. If the device RSSI is greater than -40dB, the *connectToDevice* method is called to start the connection procedure. This value is arbitrary. It can be changed to allow closer or further connections. It also depends on the transmission power set in the card reader firmware. When the connection procedure is started, scanning is stopped and the smartphone tries to connect to the device. This function is asynchronous (defined with the *async* keyword). The *await* keyword gets the promise returned by the calling function and stops the execution of the asynchronous function. When the promise is fulfilled, which happens when the promise calls the resolve function, await returns the object passed as a parameter. Processing of the function continues where it left off. The value passed as a parameter is thrown as an exception if the promise fails. This allows us to wait for a response from an asynchronous function. To call an asynchronous function, the one calling it must also be asynchronous.

In the first iteration (firmware version V1.1-without_security), the user identifier is retrieved from the Middleware using a *axios* GET method once the connection is established. This value can be written to the specific device properties. To do this, the BLE manager provides the *writeCharacteristicWithResponseForDevice* method. The device identifier, the service and the characteristic UUID are passed as parameters. The value must be base64 encoded before writing. The promise is waited for and its decoded value is compared with the one sent. If they match, the write is successful and true is returned. The device is then disconnected using the *disconnectFromDevice* function. If the device is disconnected for any reason, the list is cleared and the button must be pressed to restart a connection procedure.

Two methods have been implemented to guide and inform the user. The first is a small text printed below the button. It informs the user of the progress of the procedure. When discovery starts, *Discovering...* is printed. As soon as a device is connected, its name and address are displayed to indicate the connection. The second method is alerts. These are pop-up messages that need to be validated:

- The permissions are not granted.
- No user is selected when starting the discover.
- The authentication protocol succeed.
- The authentication protocol failed.

The application must receive the Bluetooth Low Energy notification to implement the second iteration with the authentication protocol. To enable this, the *monitorCharacteristicForService* function must be called from the connected device object. It takes three parameters, the service and the characteristic UUID. The third parameter is the

Listener. The latter is defined earlier in the code. It takes as parameter an error or a **react-native-ble-plx** characteristic object. They can be null. It checks whether an error has been received or the characteristic is empty, otherwise it stores the value in a variable. The received value must be decoded from base64 format. The listener is called each time a notification is sent on the set characteristic. It cannot be called if the program is in a while or for loop. The BLE communication is set up. The authentication protocol can now be implemented.

Authentication protocol

As for the card reader, a SM will follow the procedure and perform the authentication. The states of this machine are similar to those of the reader. They have been defined according to the authentication protocol diagram. The operations performed in each state had to be precisely described before programming the machine. The transitions were also chosen. This analysis led to the following diagram. It contains all the states, the transitions and a short explanation of the purpose of each state.

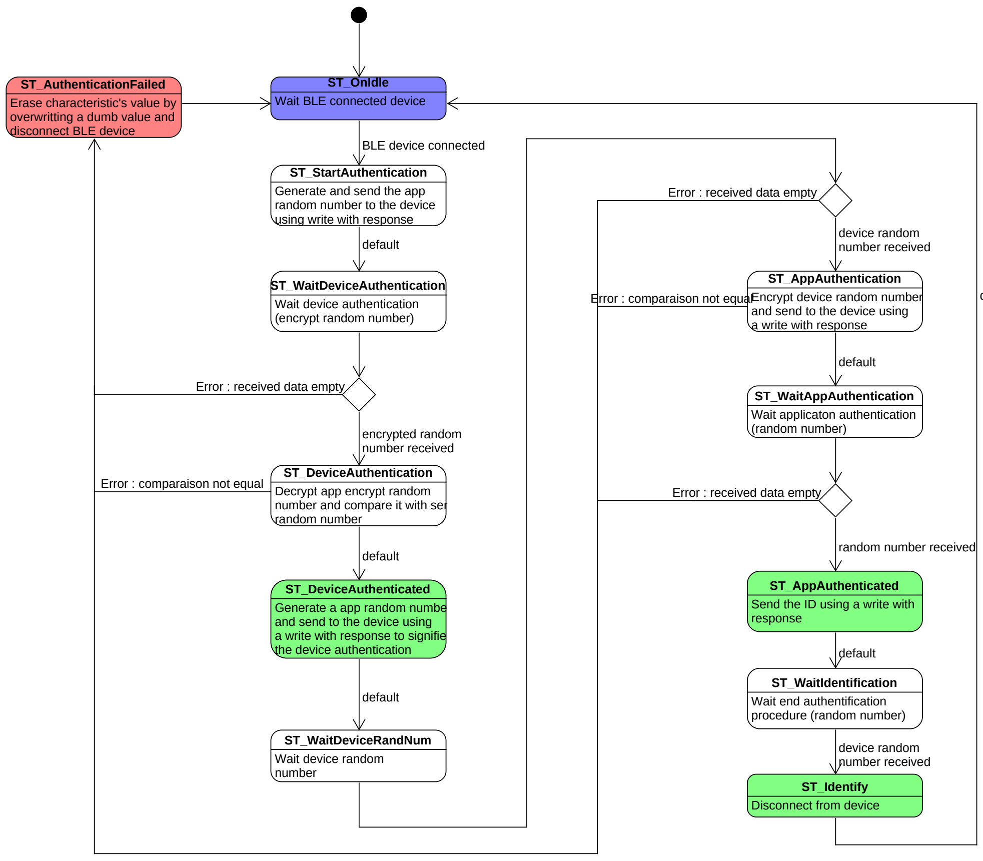

ST_AuthenticationFailed
Erase characteristic's value by overwritting a dumb value and disconnect BLE device
ST_OnIdle
Wait BLE connected device
BLE device connected
ST_StartAuthentication
Generate and send the app random number to the device using write with response
default
ST_WaitDeviceAuthentication
Wait device authentication (encrypt random number)
Error : received data empty
encrypted random number received
Error : comparaison not equal
ST_DeviceAuthentication
Decrypt app encrypt random number and compare it with ser random number
default
ST_DeviceAuthenticated
Generate a app random numbe and send to the device using a write with response to signifie the device authentication
default
ST_WaitDeviceRandNum
Wait device random number
Error : received data empty
device random number received
Error : comparaison not equal
ST_AppAuthentication
Encrypt device random number and send to the device using a write with response
default
ST_WaitAppAuthentication
Wait applicaton authentication (random number)
Error : received data empty
random number received
ST_AppAuthenticated
Send the ID using a write with response
default
ST_WaitIdentification
Wait end authentification procedure (random number)
device random number received
ST_Identify
Disconnect from device
default

Figure 5.15: Mobile application's SM diagram

The machine is in the BLE component. The *chooseSMstate* function contains the while loop for the SM. It does not continuously check the states. As a result, the notification listener cannot be called and they are not processed. Each time a notification is expected, the while loop must be exited. As soon as the notification is received, it calls back the *chooseSMstate* to restart the state machine. This is why the switch case has several *return* statements.

As soon as a device is connected, the state machine is started and goes directly into the ST_StartAuthentication state. The authentication process starts. It writes a random number in the characteristic *495f449c-fc60-4048-b53e-bdb3046d4495*. This random number is generated in the Middleware component when a GET request is made by the *getRandomNum* functions. If the value is written correctly, the state machine switches to ST_WaitDeviceAuthentication. The state machine is stopped to allow the notification to be received.

The value written is encrypted by the card reader and returned. Once the notification has been received, the machine enters the ST_WaitDeviceAuthentication state. The data received is decrypted with the *getDecryptedData* function and compared with the random value sent. If the comparison is successful, the card reader is authenticated and can proceed to the next state (ST_DeviceAuthenticated). A random number is written to the TWN4 to indicate the success of the first authentication. It then waits for the random number of the reader to be encrypted (ST_WaitDeviceRandNum). If the received value is different, the authentication has failed. The SM enters an authentication failed state (ST_AuthenticationFailed). The remote device is then disconnected and the current state is set to ST_OnIdle, ready to start a new authentication.

When the reader's random number is received, the state machine enters the ST_AppAuthentication state. It encrypts the received number with the *getEncryptedData* method, using the GET cryptography methods provided by the Middleware. This value is written to the card reader and the SM forwards the authentication confirmation in the ST_WaitAppAuthentication state.

The last step of the authentication protocol is the identification using the signed message provided by the web service. The signed message is sent to the reader when the application authentication confirmation is received and the machine waits in the ST_WaitIdentification state. If the signed message is valid, an acknowledgement is sent by the reader as a random number passed in a notification. Finally, the last state is reached (ST_Identify). A warning is displayed to inform the user and the remote device is disconnected. A new authentication can be performed from the beginning.

Whenever the writing of the characteristic or the GET request to the Middleware fails, the state machine goes to ST_AuthenticationFailed. The procedure is then stopped and the card reader is disconnected.

A sequence diagram is available in appendix. It shows a complete authentication process, including the card reader, the mobile application and the Middleware components. It also shows all the states of both state machine. This diagram helps to understand the communication between the two BLE devices and the web service.

5.5 Validation

BLE communication is now established. The card readers and the application can exchange data and authentication is possible. This section analyses the Bluetooth communication set up to check that it is working correctly. All these checks have been carried out on the basis of packets exchanged using the Bluetooth sniffer. The different logs are available in the repository.

First of all, the card reader advertises the advertising string correctly. It contains the correct data. The private service is visible. It is sent to the three primary notification channels as set during initialisation. The following figure shows the three packets for each channel: 37, 38 and 39.

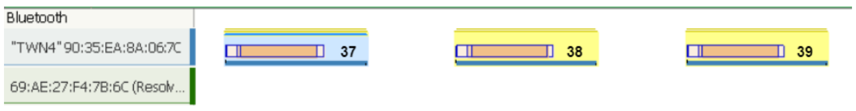
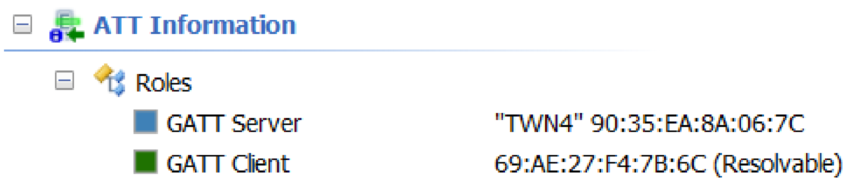


Figure 5.16: BLE advertising channel

The peripheral is then visible to all remote devices. When the connection is initiated by the application, ATT packets are exchanged to allow the connection. In the connection procedure, the Maximum Transmission Unit (MTU) is exchanged. It is 250 bytes. The sever and client can support this length. This is more than sufficient for this project. The longest message transferred is 32 bytes.

Once connected, the GATT roles are defined as shown in the figure below. This corresponds to the defined roles.



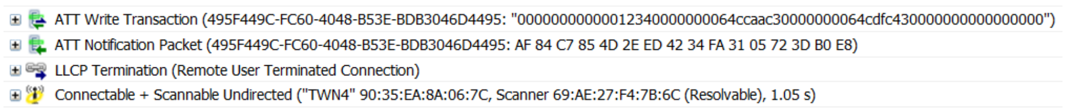
ATT Information

Roles

🟦 GATT Server	"TWN4" 90:35:EA:8A:06:7C
🟩 GATT Client	69:AE:27:F4:7B:6C (Resolvable)

Figure 5.17: BLE GATT roles

The authentication process is then started. The various data are exchanged using Write and Notify. The following figure shows the last step of the authentication protocol. The write transaction corresponds to the 32-byte signed message sent by the application. The user identifier, the current time and the expiry time can be seen. The data is in string format compared to the next transaction which is in hexadecimal. As explained above, there is no type in the application coded in JavaScript. The data is therefore stored and sent in string format. The card reader converts this data to the correct format for processing.

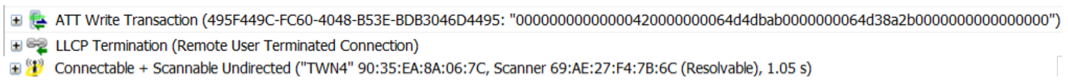


```
ATT Write Transaction (495F449C-FC60-4048-B53E-BDB3046D4495: "0000000000001234000000006 4ccaac30000000064cdfc430000000000000000")
ATT Notification Packet (495F449C-FC60-4048-B53E-BDB3046D4495: AF 84 C7 85 4D 2E ED 42 34 FA 31 05 72 3D B0 E8)
LLCP Termination (Remote User Terminated Connection)
Connectable + Scannable Undirected ("TWN4" 90:35:EA:8A:06:7C, Scanner 69:AE:27:F4:7B:6C (Resolvable), 1.05 s)
```

Figure 5.18: BLE exchange packets sunny case

The notification packet is the authentication confirmation sent by the card reader. The format is correct as sent by the card reader. The operation is then completed and the application disconnects from the peripheral. The peripheral then starts advertising again. The behaviour of the authentication sunny case is verified.

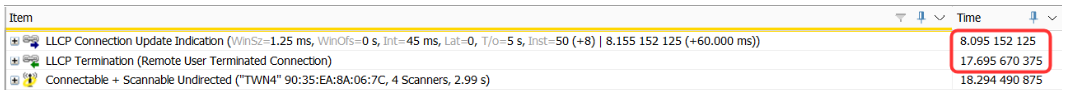
Two other tests were carried out. The connection was tried with an expired sigend message. To create this situation, the *getSignedMessage* function of the Middleware was modified. The expiration time has been set to one day in the past. Both devices can be authenticated. Therefore, when the card reader checks the validity of the signed message, the authentication fails. Compared to the previous figure, there was no notification transaction after writing the signed message as the latter is not valid. The peripheral disconnects the remote device as shown in the figure below.



```
⊞ ATT Write Transaction (495F449C-FC60-4048-B53E-BDB3046D4495: "00000000000000420000000064d4dbab0000000064d38a2b0000000000000000")
⊞ LLCP Termination (Remote User Terminated Connection)
⊞ Connectable + Scannable Undirected ("TWN4" 90:35:EA:8A:06:7C, Scanner 69:AE:27:F4:7B:6C (Resolvable), 1.05 s)
```

Figure 5.19: BLE example packets signed message expired

The final test was to try to connect to an unauthorised device. This was done using the *nRF connect* application. It can detect and connect to the TWN4. However, the procedure must be initiated by the smartphone. If nothing is done, the card reader will automatically disconnect the remote device after the ten-second timeout set for the connection.



Item	Time
⊞ LLCP Connection Update Indication (WinSz=1.25 ms, WinOfs=0 s, Int=45 ms, Lat=0, T/o=5 s, Inst=50 (+8) | 8.155 152 125 (+60.000 ms))	8.095 152 125
⊞ LLCP Termination (Remote User Terminated Connection)	17.695 670 375
⊞ Connectable + Scannable Undirected ("TWN4" 90:35:EA:8A:06:7C, 4 Scanners, 2.99 s)	18.294 490 875

Figure 5.20: BLE example packets unauthorized device

The timeout is shown in the figure above. The LLCP Connection Update item is the last step of the connection procedure. The remote device will be disconnected after 10 seconds. The system has responded correctly to the connection of unauthorised devices. BLE communication is controlled and working.

6 | Conclusion

To conclude this report, the first part of this chapter summarises the developments. The result is compare to the initial objectives. Recommendations for future work are then briefly discussed.

6.1 Summary

Bluetooth Low Energy (BLE) communication has been established between a *Elatec* TW4 card reader and a mobile application. The aim is to enable identification so that printouts can be enabled on a printer. To achieve this, the ecosystem of the industrial partner was analysed and an architecture was defined.

A Print Manager server was set up to manage printers, users and costs. It was installed on a network in NAT mode. A small Java application was used to make changes to the server using the provide API. The PaperCut Manager can be used to add or remove one or more users and check their balance.

The mobile application needs to retrieve data from the server. A middleware component connects both. It also provides cryptographic methods for the authentication protocol. This component is a web service based on the *Spring* framework. The React Native application also communicates with the card reader using BLE. An authentication protocol verifies the identity of the card reader and the application. Once authenticated, the validity of the signed message sent is checked. This excludes expired messages. The card reader firmware has been implemented to enable this connection and authentication. This connection is therefore secure thanks to the protocol.

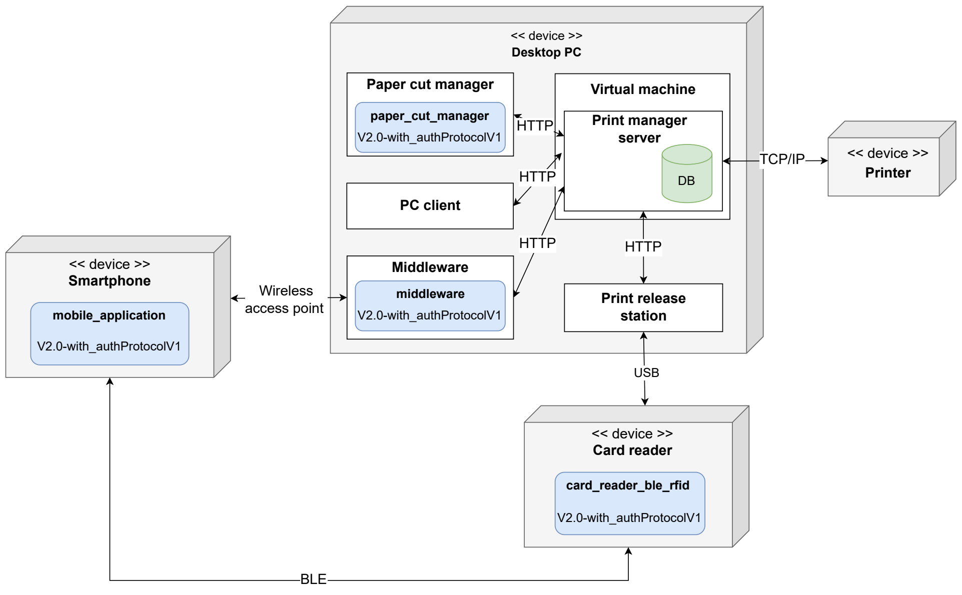


Figure 6.1: Deployment diagram

Above, the complete architecture is shown in a deployment diagram. It represent the test bench set up for this thesis.

6.2 Comparison with the objectives

To validate this Proof oc Concept, individual tests were carried out at each stage of the implementation of the various components. This validated their operation. The whole system was then tested. The final version of the architecture allows secure identification to a printer in order to release print jobs.

Five objectives have been defined in the introduction. They have guided the development of this thesis. The first objective was to analyse the existing *Polyright* identification system and ecosystem. This analysis forms the basis for the development of the various components. They have always been designed to make it as easy as possible to reuse the results of this PoC.

A complete test bench has been set up to allow testing under real conditions. The commissioning of this setup is described in the following appendix. It was essential to have a functional test bench to validate the behaviour of the system. This validates objective two.

The identification process had to be secure. To achieve this, an existing protocol was analysed as a basis. A new authentication protocol could then be specified, taking into account the needs of the industrial partner. The aim is to avoid any kind of attacks:

- ✓ **Denial of Service**: The central device will automatically be disconnected after ten seconds.
- ✓ **Eavesdropping**: The exchange message contains no useful information.
- ✓ **Replay Attack**: The message is only valid once, as new random numbers are used for each authentication attempt.
- ✗ **Man in the Middle**: Depends on the case. The MITM must replicate the reader's advertisement packet unless the application fails to connect (looking for the private service). If the connection is made, the data can be modified. Authentication will fail if an authentication packet is altered. However, a false user identifier could be transmitted if only the signed message is tampered with.
- ✗ **Relay Attack**: If unmodified data is relayed, authentication can be performed.

Objective three is only partially achieved. Attacks can cause problems despite the authentication protocol.

The last two objectives involved setting up Bluetooth Low Energy communication for both the card reader and the application. For this purpose, the firmware of the card reader and a test application were developed. These two objectives were then acquired.

6.3 Problems encountered

The absence of a debugger for the card reader has slowed down the firmware implementation considerably. This had to be done via a printed message or by modifying the Generic Attribute profile (GATT) profile (can be analysed with the sniffer or *nRF connect* application). The time required to identify, locate and fix an error has increased. Understanding the operation of the functions provided by the API was also more complicated.

The security module could not be used because the card reader's bootloader could not be updated. The encryption key value had to be stored directly in the TWN4 firmware. This raises security concerns as the integrity of the key cannot be guaranteed.

The mobile application presented some problems. Using the BLE library with *Expo* turned out to be more complex than expected. The lack of documentation did not help to solve the problems. Therefore *Expo* was abandoned for the rest of the project. Using this new technology, which is React Native, took time and required research. However, it would have been the same with other technologies such as *Android Studio*.

As the type does not exist in JavaScript (in the mobile application), the hexadecimal value had to be stored as a string. This caused format problems during Bluetooth communication (0x1, 0x2, 0x3, 0x4, ... instead of 0x12, 0x34, ...). The received values therefore had to be transformed to the correct format.

One option would have been to allow authentication via the smartphone without a connection. An old signed message could have been reused for this, provided that the expiration date was not exceeded. Unfortunately, the chosen architecture does not allow this functionality. As encryption and decryption are handled in the Middleware, a disconnected phone could not pass the authentication process. New random numbers are used for each attempt. Old encrypted messages could not be reused.

Finally, the *Print Release Station* could not be installed on the Raspberry PI. This would have provided a more realistic test bed. However, the latter remains functional thanks to the Print Release application.

6.4 Future improvements

The current version is operational, but the authentication protocol could be improved. The procedure specified and implemented is a simple but secure version. As seen before, the system is not protected against all types of attacks. New security mechanisms need to be added to prevent Man In The Middle and Relay Attacks. One solution would be to add a mechanism to verify the identity of devices during the connection establishment process. This could include verifying device certificates or using device-specific identifiers. This will avoid connecting to an unauthorised device.

The Bluetooth module of the card reader using BLE 4.2 could also use the embedded BLE encryption. The devices must be configured to exchange the encryption key. A *Passkey Entry* could be used, but requires the passkey to be known to the users. All messages would then be encrypted, reducing the risk of eavesdropping.

To increase the level of security, it would be necessary to use card readers with a bootloader that allows the use of the security module (above V1.20). As explained, there are better solutions than a shared key in terms of security. It would be better to use a private/public key combination.

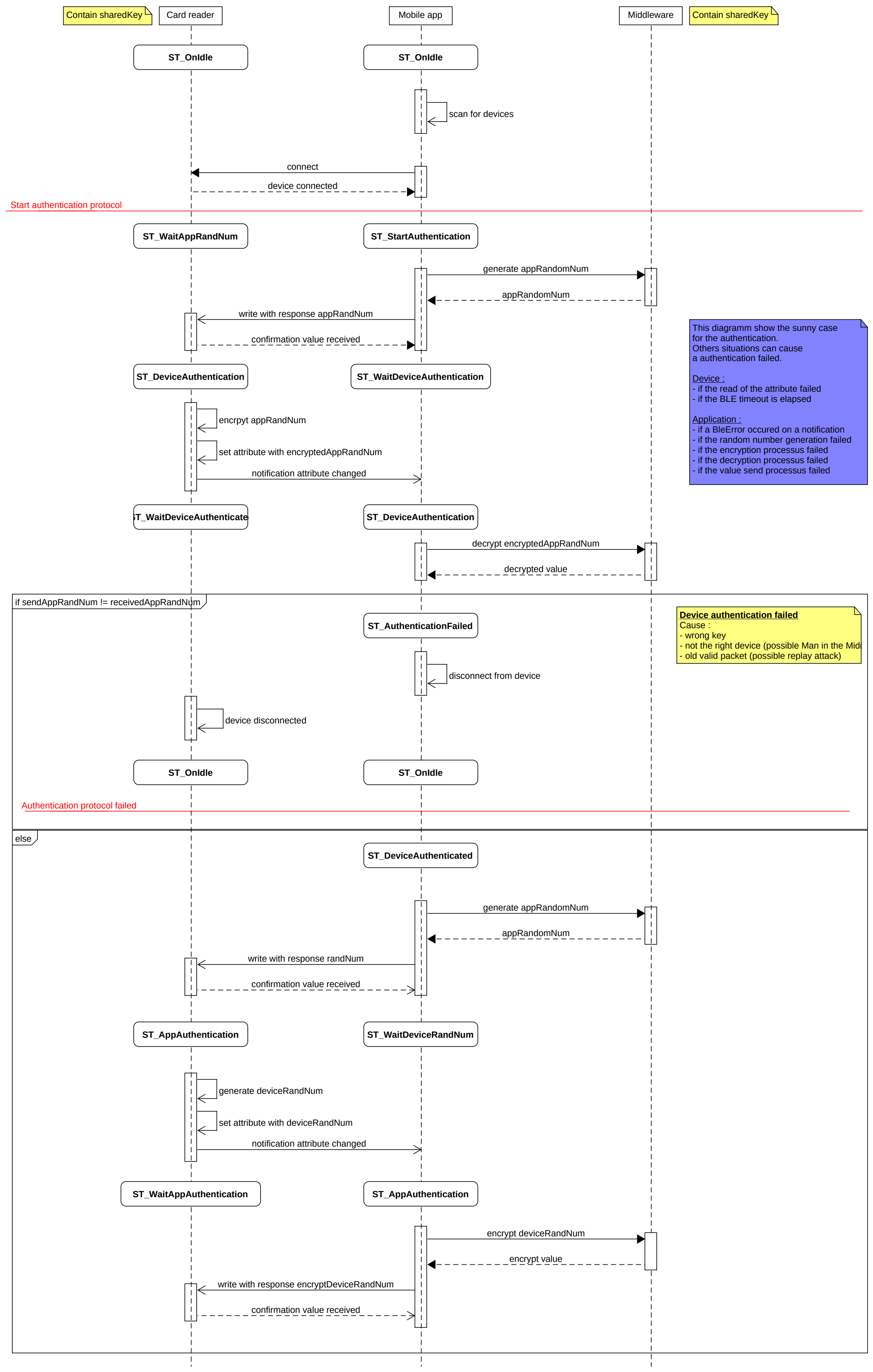
Finally, if authentication is to be done offline, the security mechanism must be moved to the mobile application. It is also possible to redefine a new authentication protocol to allow this functionality.

Sion, 18th August 2023
Adrien Rey

A | Authentication sequence

This diagram represents the sunny case of the authentication protocol including the card reader, the mobile application and the Middleware. It is shows on two pages.

Appendix A. Authentication sequence

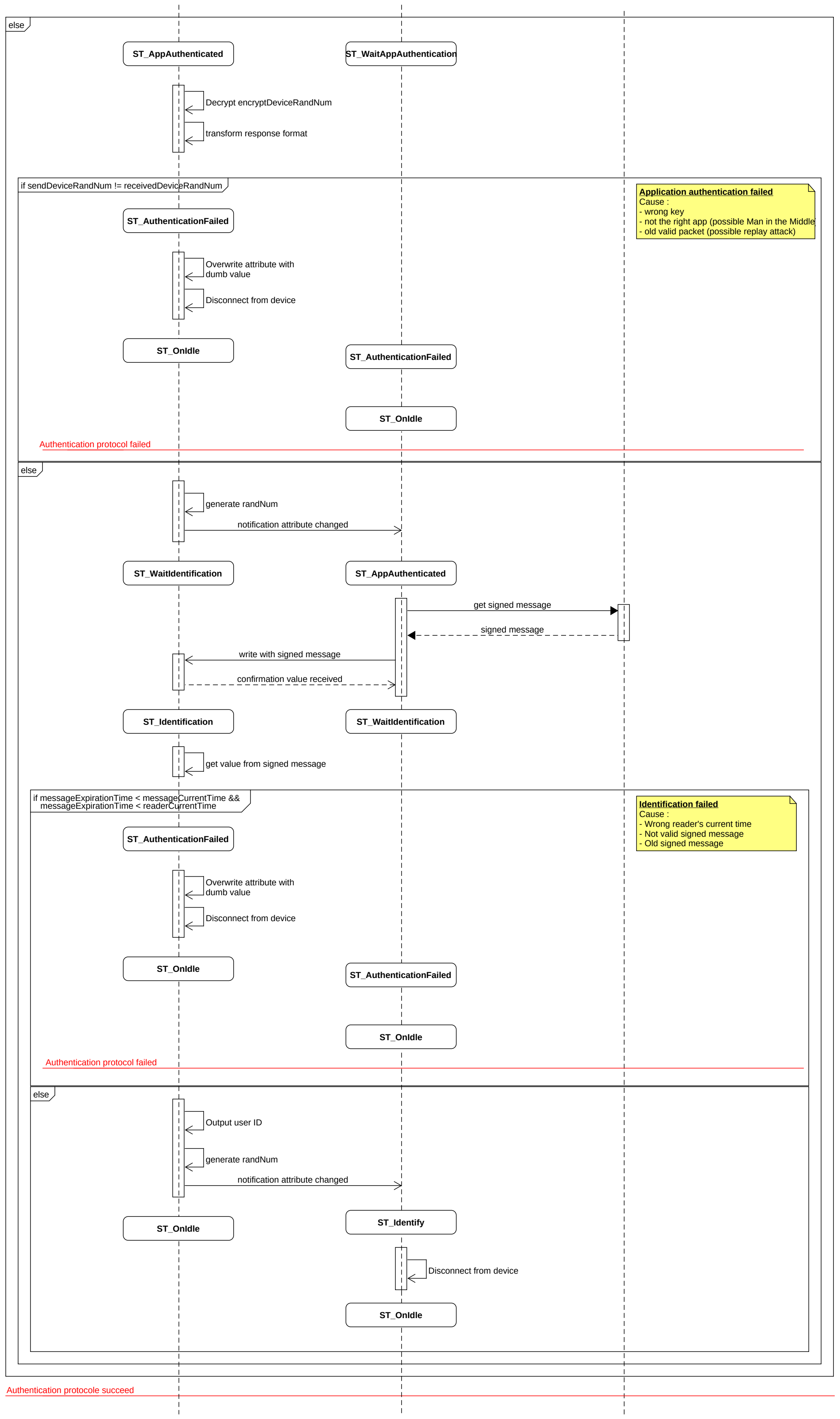


This diagramm show the sunny case for the authentication.
Others situations can cause a authentication failed.

Device :
- if the read of the attribute failed
- if the BLE timeout is elapsed

Application :
- if a BleError occured on a notification
- if the random number generation failed
- if the encryption processus failed
- if the decryption processus failed
- if the value send processus failed

Device authentication failed
Cause :
- wrong key
- not the right device (possible Man in the Mid
- old valid packet (possible replay attack)


else
ST_AppAuthenticated
ST_WaitAppAuthentication
Decrypt encryptDeviceRandNum
transform response format
if sendDeviceRandNum != receivedDeviceRandNum
Application authentication failed
Cause :
- wrong key
- not the right app (possible Man in the Middle
- old valid packet (possible replay attack)
ST_AuthenticationFailed
Overwrite attribute with dumb value
Disconnect from device
ST_OnIdle
ST_AuthenticationFailed
ST_OnIdle
Authentication protocol failed
else
generate randNum
notification attribute changed
ST_WaitIdentification
ST_AppAuthenticated
get signed message
signed message
write with signed message
confirmation value received
ST_Identification
ST_WaitIdentification
get value from signed message
if messageExpirationTime < messageCurrentTime && messageExpirationTime < readerCurrentTime
Identification failed
Cause :
- Wrong reader's current time
- Not valid signed message
- Old signed message
ST_AuthenticationFailed
Overwrite attribute with dumb value
Disconnect from device
ST_OnIdle
ST_AuthenticationFailed
ST_OnIdle
Authentication protocol failed
else
Output user ID
generate randNum
notification attribute changed
ST_OnIdle
ST_Identify
Disconnect from device
ST_OnIdle
Authentication protocole succeed

B | Development environment

The full source codes of the thesis, as well as a copy of the report, are available online on the following *Git* repository.

- NetPrinting_Identification_Prog
https://github.com/HEI-TB-Rey/NetPrinting_Identification_Prog

The tools and libraries used for this thesis are listed in the following sections. All the Java components use Java Development Kit 20.0.1.

B.1 Middleware (Java)

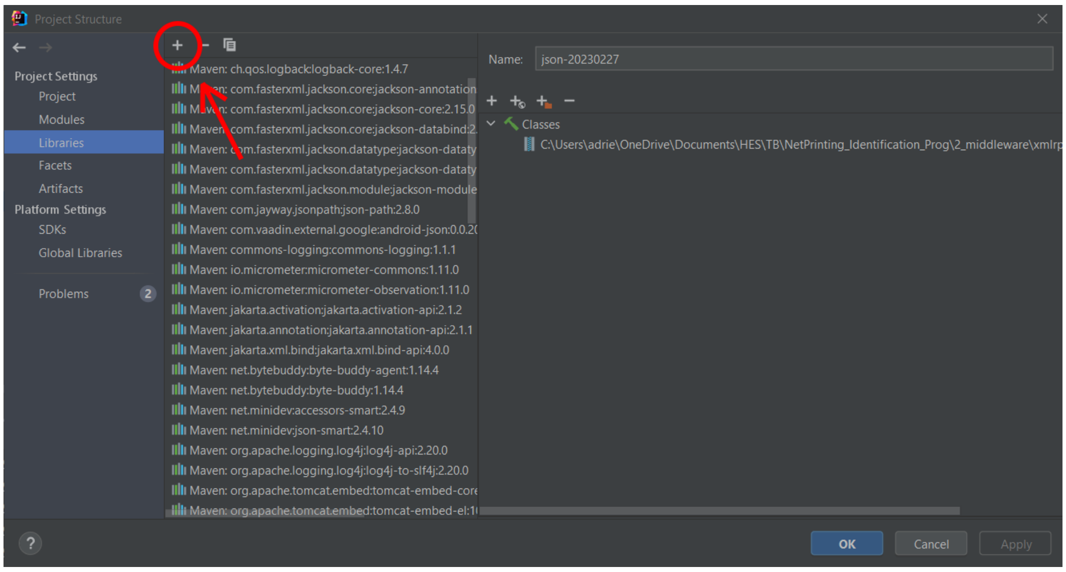
The followings tool and libraries have been used to develop this component.

Tools	**Version**	**Website**
IntelliJ	2021.3	https://www.jetbrains.com/idea/download/
Apache Maven	3.9.2	https://maven.apache.org/download.cgi

Libraries	**Version**	**Website**
xmlrpc	2.0	https://mvnrepository.com/artifact/xmlrpc/xmlrpc
commons-codec	1.3	https://mvnrepository.com/artifact/commons-codec/commons-codec/1.3

Table B.1: Used tools and libraries in the Middleware

These libraries are included in the Middleware source code. The following procedure must be followed if they need to be added. Once the .jar library files have been downloaded and placed in a project folder, they can be added to the project via File->Project Structure->Libraries in *IntelliJ*. The following panel will open.



```
Maven: ch.qos.logback:logback-core:1.4.7
Maven: com.fasterxml.jackson.core:jackson-annotation
Maven: com.fasterxml.jackson.core:jackson-core:2.15.0
Maven: com.fasterxml.jackson.core:jackson-databind:2.
Maven: com.fasterxml.jackson.datatype:jackson-dataty
Maven: com.fasterxml.jackson.datatype:jackson-dataty
Maven: com.fasterxml.jackson.module:jackson-module
Maven: com.jayway.jsonpath:json-path:2.8.0
Maven: com.vaadin.external.google:android-json:0.0.20
Maven: commons-logging:commons-logging:1.1.1
Maven: io.micrometer:micrometer-commons:1.11.0
Maven: io.micrometer:micrometer-observation:1.11.0
Maven: jakarta.activation:jakarta.activation-api:2.1.2
Maven: jakarta.annotation:jakarta.annotation-api:2.1.1
Maven: jakarta.xml.bind:jakarta.xml.bind-api:4.0.0
Maven: net.bytebuddy:byte-buddy-agent:1.14.4
Maven: net.bytebuddy:byte-buddy:1.14.4
Maven: net.minidev:accessors-smart:2.4.9
Maven: net.minidev:json-smart:2.4.10
Maven: org.apache.logging.log4j:log4j-api:2.20.0
Maven: org.apache.logging.log4j:log4j-to-slf4j:2.20.0
Maven: org.apache.tomcat.embed:tomcat-embed-core
Maven: org.apache.tomcat.embed:tomcat-embed-el:1
```

Name: json-20230227

C:\Users\adrie\OneDrive\Documents\HES\TB\NetPrinting_Identification_Prog\2_middleware\xmlrp

Figure B.1: Adding libraries to IntelliJ

Once the plus button is pressed, select Java and browse for the library to add. It will then be added to the project. The *pom.xml* file needs to be modified to include these dependencies by adding the following code under *<dependencies>*.

```
<dependency>
    <groupId>xmlrpc</groupId>
    <artifactId>xmlrpc</artifactId>
    <version>2.0</version>
</dependency>

<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.3</version>
</dependency>
```

Listing B.1: Code - Dependencies to add in the pom file

The Middleware project can be then open and run.

B.2 PaperCut manager (Java)

The PaperCut manager required the following tools and libraries.

Tools	Version	Website
IntelliJ	2021.3	https://www.jetbrains.com/idea/download/
Apache Maven	3.9.2	https://maven.apache.org/download.cgi

Libraries	Version	Website
xmlrpc	2.0	https://mvnrepository.com/artifact/xmlrpc/xmlrpc
commons-codec	1.3	https://mvnrepository.com/artifact/commons-codec/commons-codec/1.3

Table B.2: Used tools and libraries in the PaperCut manager

The libraries are included in the project. The procedure for adding the libraries is the same as in the previous section if you need to add these libraries.

B.3 Card reader (C)

The followings tool and libraries have been used to develop the card reader firmware.

Tools	Version	Website
Visual Studio Code	-	https://code.visualstudio.com/
TWN4 DevPack	4.51	https://www.elatec-rfid.com/int/elatec-software

Table B.3: Used tools for the card reader firmware

The development pack contain the *AppBlaster* tool to create and load the firmware's image and all the documentations.

B.4 Mobile application (React Native)

The mobile application required the following tools and libraries.

Tools	Version	Website
Visual Studio Code	-	https://code.visualstudio.com/
Node.js	18.61.1	https://nodejs.org/fr
npm	9.5.1	include in Node.js
Libraries	**Version**	**Website**
react	18.2.0	https://react.dev/learn/installation
react-native	0.72.0	https://reactnative.dev/docs/environment-setup
react-native-ble-plx	2.0.3	https://www.npmjs.com/package/react-native-ble-plx
react-native-base64	0.2.1	https://www.npmjs.com/package/react-native-base64
react-native-permissions	3.8.1	https://www.npmjs.com/package/react-native-permissions
axios	1.4.0	https://www.npmjs.com/package/axios
buffer	6.0.3	https://www.npmjs.com/package/buffer

Table B.4: Used tools and libraries for the mobile application

Node.js has to be downloaded. The node module allows to run the the application on the development smartphone. This module is not in the repository and must be downloaded.

The *react* and *react-native* libraries are added when the project is created. All other libraries are already included in the project. They can be added to the project folder using the *npm* command available on the corresponding website.

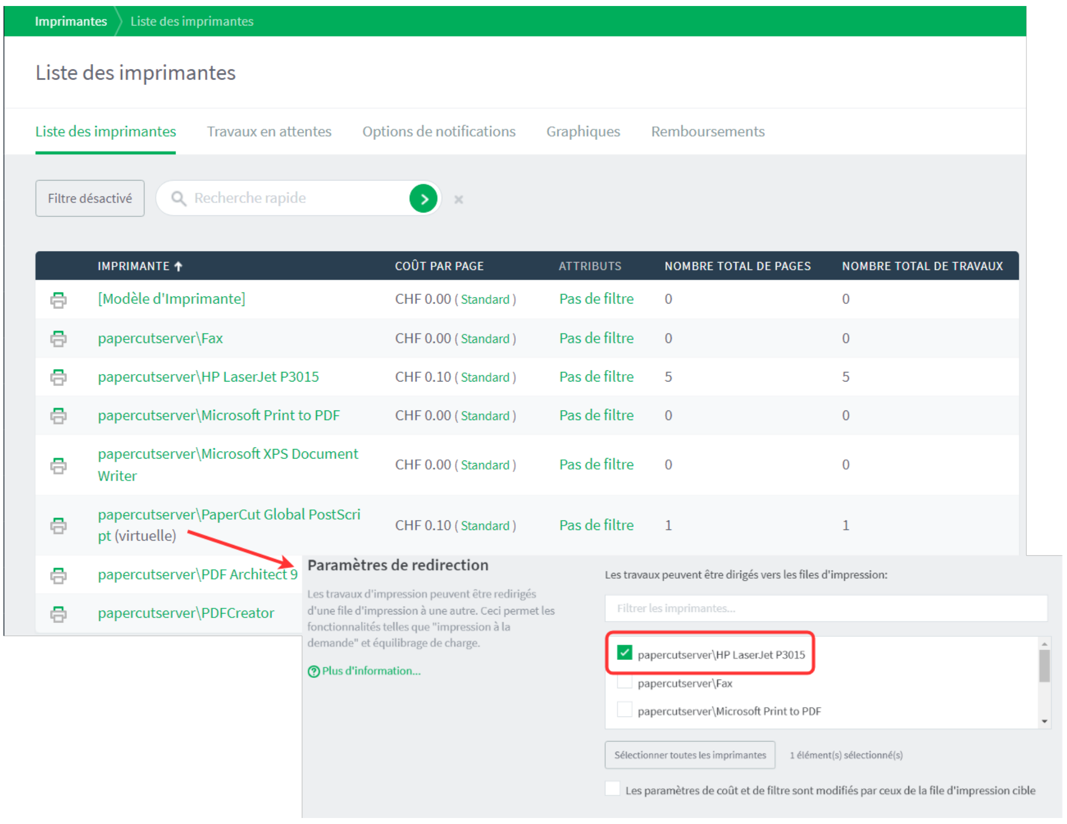
Setup is complete and the application can be run following the instructions in the **React Native CLI Quickstart** guide [31]. A Google Pixel 3 XL smartphone was used for testing, but other Android phones can be used in developer mode. The application has not been tested on iOS smartphones. Behaviour is therefore not guaranteed.

C | Commissioning of the test bench

This appendix describes the procedure for setting up the test bench. This procedure starts the test bench using the codes provided in the Git repository (`https://github.com/HEI-TB-Rey/NetPrinting_Identification_Prog`). It is not a procedure for starting the test bench from scratch.

1. **The virtual machine (VM) containing the Print Manager server can be opened using VMware Workstation Player.**
   The server will start automatically. Use *http://papercutserver:9191/admin* to access the server's web service (username: *admin*, password: *admin1*)

2. **If the printer is not connected, it must be added to the VM and linked to the Null Port.**
   It can be controlled in the Print Manager web service. Once the printer has been added to the virtual machine, it will appear under the Print Manager printers in the web service. The new printer can be added to the Null Port printer's linked printers list. An HP LaserJet P3015 printer has been provided for this thesis. See the Print queue section to add this particular printer.



IMPRIMANTE	COÛT PAR PAGE	ATTRIBUTS	NOMBRE TOTAL DE PAGES	NOMBRE TOTAL DE TRAVAUX
[Modèle d'Imprimante]	CHF 0.00 (Standard)	Pas de filtre	0	0
papercutserver\Fax	CHF 0.00 (Standard)	Pas de filtre	0	0
papercutserver\HP LaserJet P3015	CHF 0.10 (Standard)	Pas de filtre	5	5
papercutserver\Microsoft Print to PDF	CHF 0.00 (Standard)	Pas de filtre	0	0
papercutserver\Microsoft XPS Document Writer	CHF 0.00 (Standard)	Pas de filtre	0	0
papercutserver\PaperCut Global PostScript (virtuelle)	CHF 0.10 (Standard)	Pas de filtre	1	1
papercutserver\PDF Architect 9				
papercutserver\PDFCreator				

Paramètres de redirection

Les travaux d'impression peuvent être redirigés d'une file d'impression à une autre. Ceci permet les fonctionnalités telles que "impression à la demande" et équilibrage de charge.

Plus d'information...

Les travaux peuvent être dirigés vers les files d'impression:

Filtrer les imprimantes...

- ✓ papercutserver\HP LaserJet P3015
- papercutserver\Fax
- papercutserver\Microsoft Print to PDF

Sélectionner toutes les imprimantes — 1 élément(s) sélectionné(s)

Les paramètres de coût et de filtre sont modifiés par ceux de la file d'impression cible

Figure C.1: *Link printer in the web service*

3. **The Print Release Station must be started to allow the print job to be released.**
   The *connection.properties*, on the VM (*C:/Program Files/PaperCut MF/release)*, must be modified with the IP address of the server. This can be obtained by running *ping -4 papercutserver* at a command prompt.

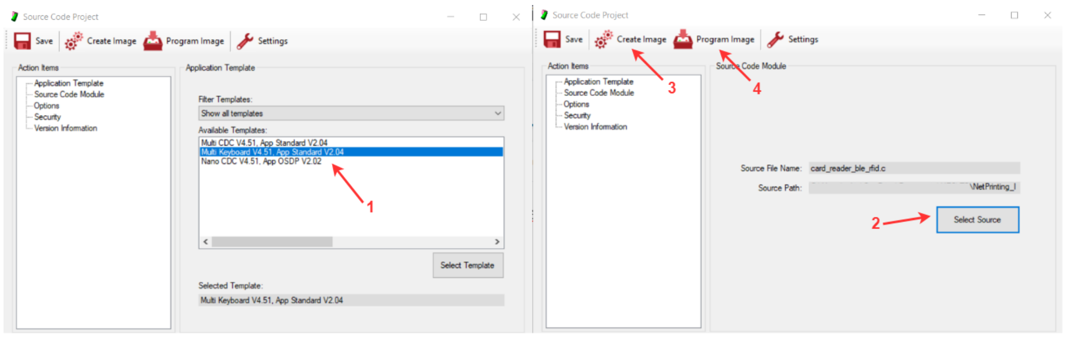
```
#Bootstrap configuration information
server-name=papercutserver
server-ip=192.168.100.140
server-port=9191
```

   Listing C.1: Code - Example configuration server information

   The application can then be found on the shared network in the *PaperCut MF/release* folder. It opens in full screen and cannot be closed. The only way to close it is to log on with the administrator account.

4. **The PC Client application can be launched on the host computer.**
   It allows the *print as* function. It can be started under the shared network *//PAPERCUTSERVER/PCClient/win*. The *config.properties*, on the VM (*C:/Program Files/PaperCut MF/client/win/*), must be modified with the same IP address as in the point 3. The admin account can be used to log in.

5. **The card reader can be connected to the host computer.**
   The image must be created from the *card_reader_ble_rfid.c* file using the *AppBlaster* application.



Selected Template: Multi Keyboard V4.51, App Standard V2.04

Source File Name: card_reader_ble_rfid.c

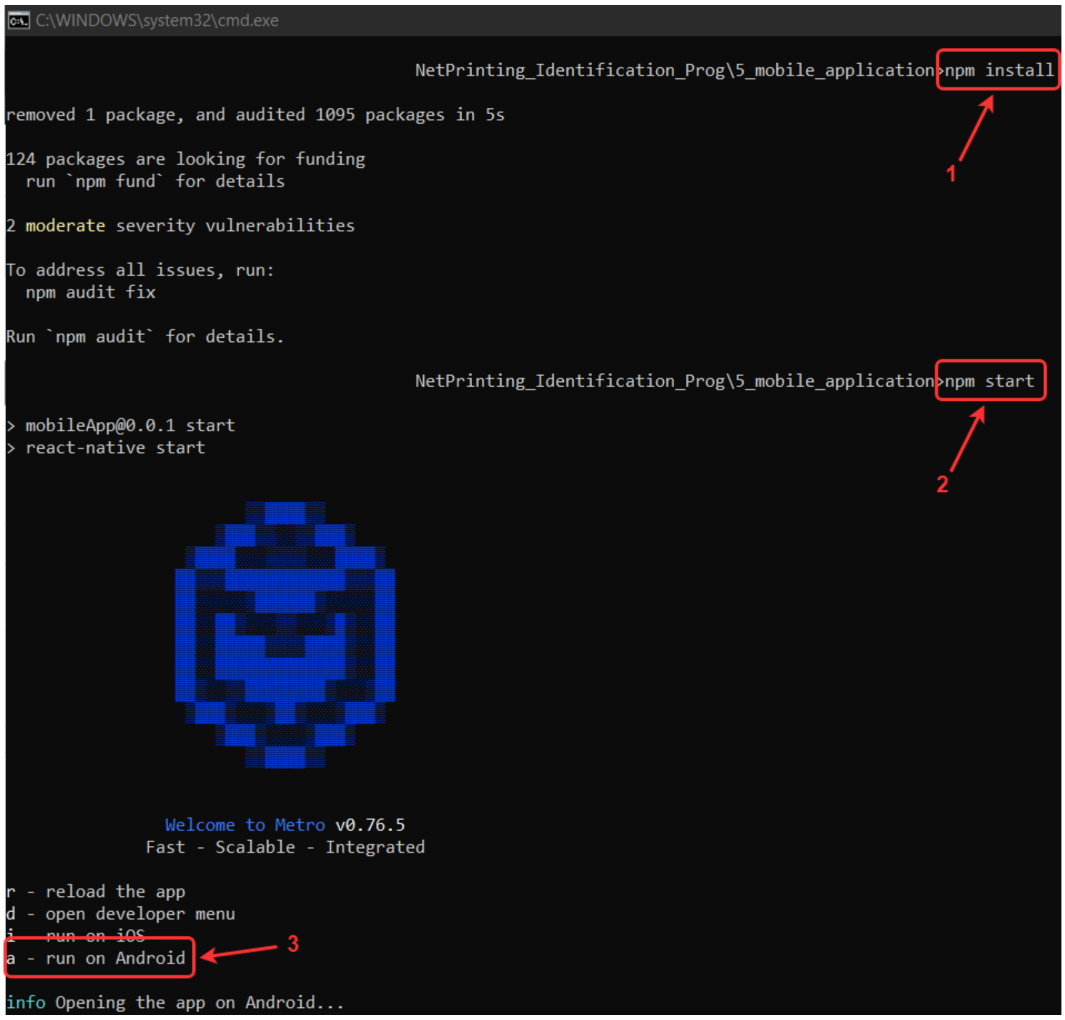
Source Path: \NetPrinting_I

Figure C.2: Load firmware into the card reader

6. **The Middleware component can be launched from the IDE or from the generated .jar file.**

7. **A mobile wireless access point must be started on the host computer.**
   The phone needs to connect to it.

8. **The mobile application can be loaded onto the development phone.**
After opening a prompt in the mobile application folder, the following procedure can be followed with the phone connected to the computer.



```
                                          NetPrinting_Identification_Prog\5_mobile_application>npm install

removed 1 package, and audited 1095 packages in 5s

124 packages are looking for funding
  run `npm fund` for details

2 moderate severity vulnerabilities

To address all issues, run:
  npm audit fix

Run `npm audit` for details.

                                          NetPrinting_Identification_Prog\5_mobile_application>npm start

> mobileApp@0.0.1 start
> react-native start


               Welcome to Metro v0.76.5
             Fast - Scalable - Integrated

r - reload the app
d - open developer menu
i - run on iOS
a - run on Android

info Opening the app on Android...
```

Figure C.3: *Open mobile application on development smartphone*

9. **The PaperCut manager .jar can be opened.**
Modifications should then be made on the Print Manager server. Users can be added or deleted. The balance can also be changed.

The test bench is set up. When a user sends a print job from the host computer, the PC Client asks for the user's name. It is then sent to the print queue. Once identified with a card or the mobile application at the Print Release Station, the job is released and the document is printed.

D | Impact of bachelor thesis on sustainability

This chapter aims to analyse the impact of this bachelor thesis on sustainable development. Sustainable development is a central issue today. It must influence our decisions to adopt a long-term perspective and to integrate environmental and social constraints into the economy. This term was first defined in the 1987 *Brundtland Report* as "*development that meets the needs of the present without compromising the ability of future generations to meet their own needs*".[1] In 2015, the UN defined 17 goals, including education, energy, industrialisation or biodiversity, to achieve sustainable development.[2] They are called the **Sustainable Development Goals (SDG)**, and most of these goals will be achieved by 2030.

SUSTAINABLE DEVELOPMENT GOALS
1 NO POVERTY
2 ZERO HUNGER
3 GOOD HEALTH AND WELL-BEING
4 QUALITY EDUCATION
5 GENDER EQUALITY
6 CLEAN WATER AND SANITATION
7 AFFORDABLE AND CLEAN ENERGY
8 DECENT WORK AND ECONOMIC GROWTH
9 INDUSTRY, INNOVATION AND INFRASTRUCTURE
10 REDUCED INEQUALITIES
11 SUSTAINABLE CITIES AND COMMUNITIES
12 RESPONSIBLE CONSUMPTION AND PRODUCTION
13 CLIMATE ACTION
14 LIFE BELOW WATER
15 LIFE ON LAND
16 PEACE, JUSTICE AND STRONG INSTITUTIONS
17 PARTNERSHIPS FOR THE GOALS

Figure D.1: Sustainable Development Goals
(Source: UN Sustainable Development Goals communications materials. ©UN)

The aim of this bachelor's thesis is to establish secure Bluetooth Low Energy (BLE) communication between a smartphone application and a printer's RFID card reader. The proposed solution intends to enable the digitalisation of RFID cards for the Swiss company *Polyright*. *Polyright* offers centralized identification, access, and payment systems through RFID cards.

[1] **Report of the World Commission on Environment and Development: Our Common Future. 1987**, **p.16**.
[2] **Source: www.un.org/sustainabledevelopment/**.

This subject is particularly related to three of the seventeen goals :

- **Goal 9**: Build resilient infrastructure, promote sustainable industrialization and foster innovation
- **Goal 10**: Reducing inequalities and ensuring no one is left behind
- **Goal 12** : Ensure sustainable consumption and production patterns

Many RFID cards are used every year and therefore thrown away. In Switzerland, more than seven million credit cards will be used in 2022, and this number is expected to grow.[3] These statistics do not include loyalty, student or transport cards. Many of these cards could be replaced by a virtual version on a smartphone application, saving plastic. This is not the best sustainable solution, but a large part of the population already uses a smartphone, and digitalising the card is not intended to make people buy new smartphones for this purpose, but rather to add functionality to existing smartphones. In addition, no new installations such as servers or devices will be set up for this digitalisation solution. Existing installations will be updated and used. This reduces the impact of this work by avoiding the creation of new costly installations in terms of materials and energy.

On the other hand, global digitalisation increases inequalities for people who do not have access to technology or the Internet. One target of Goal 9 is to increase access to communications technology and the Internet in the least developed countries. By creating a technology gap between populations, you are serving something other than this UN Sustainable Development Goal.

The project will not have a direct impact on the use of plastic cards, as this solution is dedicated to the *Polyright* environment, which is small on a global scale. However, this project is part of a movement to reduce the use of plastic and, as it is non-confidential, it could provide a basis for other companies to digitalise their system. A new solution is being developed using existing infrastructure, which limits the environmental impact. But it creates a gap with the less developed populations, but these solutions will be accessible once access to communication technologies and the Internet has increased. In conclusion, this thesis is part, in its scale, of an approach that aims to achieve sustainable development.

[3]**Source: Statista.com**, **Number of credit cards used in Switzerland from 2013 to 2028**, **2023**.

Bibliography

[1] Aurel Stevens and comparis.ch. *Etude smartphone 2020*. Nov. 2020. URL: https://fr.comparis.ch/-/media/images%202nd%20level%20page/download-center/smartphone-report-2020/comparis_smartphonestudie_2020_fr.pdf?la=fr-ch (visited on 05/30/2023).

[2] Evgeny Kalinin et al. "IoT Security Mechanisms in the Example of BLE". In: *Computers* 10.12 (Dec. 2021). Number: 12 Publisher: Multidisciplinary Digital Publishing Institute, p. 162. ISSN: 2073-431X. DOI: 10.3390/computers10120162. URL: https://www.mdpi.com/2073-431X/10/12/162 (visited on 05/24/2023).

[3] *Swagger UI*. URL: https://platform.polyright.com/api/index.html?urls.primaryName=Platform%20API (visited on 07/06/2023).

[4] *Host-based card emulation overview*. Android Developers. URL: https://developer.android.com/guide/topics/connectivity/nfc/hce (visited on 07/21/2023).

[5] *Core NFC*. Apple Developer Documentation. URL: https://developer.apple.com/documentation/corenfc (visited on 07/21/2023).

[6] *KleverKey – digital Access Control for SMB - KleverKey*. URL: https://www.kleverkey.com/en/ (visited on 07/21/2023).

[7] *Getting Started | Building an Application with Spring Boot*. Getting Started | Building an Application with Spring Boot. URL: https://spring.io/guides/gs/spring-boot/ (visited on 07/07/2023).

[8] Daniel Friyia. *friyiajr/BluetoothLowEnergySample*. original-date: 2022-08-01T18:01:21Z. Aug. 1, 2023. URL: https://github.com/friyiajr/BluetoothLowEnergySample (visited on 08/14/2023).

[9] *PaperCut MF | PaperCut*. URL: https://www.papercut.com/products/mf/ (visited on 05/25/2023).

[10] *The XML Web Services API | PaperCut*. URL: https://www.papercut.com/help/manuals/ng-mf/common/tools-web-services/ (visited on 06/06/2023).

[11] *Server commands (server-command) | PaperCut*. URL: https://www.papercut.com/help/manuals/ng-mf/common/tools-server-command/ (visited on 06/08/2023).

[12] *Release Station interfaces | PaperCut*. URL: https://www.papercut.com/help/manuals/ng-mf/releasestation/interfaces/ (visited on 05/25/2023).

[13] *Quick install: Windows | PaperCut*. URL: https://www.papercut.com/help/manuals/ng-mf/common/install-win/ (visited on 05/25/2023).

[14] *How to Setup a Nul Port on Windows | PaperCut*. URL: https://www.papercut.com/kb/Main/SetupNulPortOnWindows (visited on 08/09/2023).

[15] Letha Hughes Etzkorn. *Introduction to Middleware: Web Services, Object Components, and Cloud Computing*. Google-Books-ID: AZgnDwAAQBAJ. CRC Press, June 12, 2017. 796 pp. ISBN: 978-1-4987-5410-1.

[16] Elaine Barker. *Recommendation for Key Management Part 1: General*. NIST SP 800-57pt1r4. National Institute of Standards and Technology, Jan. 2016, NIST SP 800–57pt1r4. DOI: 10.6028/NIST.SP.800-57pt1r4. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf (visited on 07/10/2023).

[17] *The Bluetooth LE Security Study Guide*. Bluetooth® Technology Website. Oct. 25, 2019. URL: https://www.bluetooth.com/bluetooth-resources/le-security-study-guide/ (visited on 07/10/2023).

[18] Muhammad Usama Bin Aftab. *Building Bluetooth Low Energy Systems*. Packt Publishing, 2017. ISBN: 978-1-78646-108-7. URL: https://univ.scholarvox.com/book/88842767 (visited on 05/24/2023).

[19] *Assigned Numbers*. Bluetooth® Technology Website. URL: https://www.bluetooth.com/specifications/assigned-numbers/ (visited on 05/22/2023).

[20] *Bluetooth® Technology Website – The official website for the Bluetooth wireless technology. Get up to date specifications, news, and development info.* URL: https://www.bluetooth.com/ (visited on 05/22/2023).

[21] Elatec GmbH. "Mobile Apps for Android and iOS". In: (Apr. 29, 2022).

[22] Elatec GmbH. "BLE Credential Apps User Guide". In: (Apr. 29, 2022).

[23] Elatec GmbH. "BLE Implementer's Guide". In: (Apr. 29, 2022).

[24] Elatec GmbH. "Technical Handbook". In: (Apr. 29, 2022).

[25] *Download TWN4 DevPacks*. URL: https://www.elatec-rfid.com/int/twn4-dev-pack (visited on 05/17/2023).

[26] Elatec GmbH. "AppBlaster User Guide". In: (Apr. 29, 2022).

[27] Elatec GmbH. "Director User Guide". In: (Apr. 29, 2022).

[28] Elatec GmbH. *API Reference*. Apr. 29, 2022.

[29] Elatec GmbH. *Device Security*. Apr. 29, 2022.

[30] *react-native-ble-plx*. npm. Nov. 4, 2021. URL: https://www.npmjs.com/package/react-native-ble-plx (visited on 08/08/2023).

[31] *Setting up the development environment · React Native*. June 21, 2023. URL: https://reactnative.dev/docs/environment-setup (visited on 08/09/2023).

Acronyms

ATT	Attribute protocole. 29
BLE	Bluetooth Low Energy. 2, 27, 40, 59
CBC	Ciphered Block Chaining. 43
DES	Data Encryption Standard. 42
GAP	Generic Access Profile. 28
GATT	Generic Attribute profile. 28, 61
HCE	Host Card Emulation. 7
HF	High Frequency. 27
IV	Init Vector. 43
JSX	JavaScript XML. 51
LF	Low Frequency. 27
LLCP	Link Layer Connection Protocol. 33
MFD	Multifunctional Divice. 2
MFP	Multifunctional Printer. 2
MTU	Maximum Transmission Unit. 57
NFC	Near Field Communication. 7
NIST	National Institute of Standarts and Technology. 20
PoC	Proof of Concept. 3
RFID	Radio Frequency Identification. vii, 2
RSSI	Received Signal Strength Indication. 32

SM	State Machine. 45
TWN4	*Elatec* card reader. 10
UI	User Interface. 49
VM	Virtual Machine. 13

Glossary

.Net 6 .NET is a software framework developed by Microsoft that provides a programming and runtime environment for building and running various types of applications. It supports multiple programming languages, including C sharp, Visual Basic, and F sharp, allowing developers to choose their preferred language while leveraging the capabilities of the .NET platform. 20

AES AES (Advanced Encryption Standard) is a symmetric encryption algorithm that is widely used to secure sensitive data. It is a specification for the encryption and decryption of electronic data. 20

API API stands for Application Programming Interface. It is a set of rules and protocols that allows different software applications to communicate with each other. APIs define the methods, data formats, and conventions that developers should follow when interacting with a particular software component, service, or system. 6

Azure Azure is a cloud computing platform and a set of cloud services provided by Microsoft. It enables individuals and organizations to build, deploy, and manage applications and services through Microsoft's global network of data centers. 6

handle A handle is a numeric identifier assigned to each characteristic or descriptor in a GATT server. They are local to the server and are not standardized. Therefore, the same attribute may have different handle values on different devices. 29

IIS IIS (Internet Information Services) is a web server software developed by Microsoft for hosting and serving web applications and websites on Windows-based systems. It is an integral part of the Windows Server operating system and is available in different versions depending on the Windows Server edition. 6, 20

Java Java is a high-level, general-purpose programming language. It is designed to be platform-independent, meaning that Java programs can run on any device or operating system that has a Java Virtual Machine (JVM) installed. 20

JSON JSON (JavaScript Object Notation) is a lightweight data-interchange format commonly used for storing and transmitting data between a server and a client, or between different parts of an application. It is language-independent and easy for humans to read and write, while also being easily parsed and generated by machines. 20, 25

Maven Maven provides a structured way to manage a project's build process, dependencies, and documentation for Java-based projects. Maven follows the concept of convention over configuration, which means it encourages developers to follow a standard project structure and naming conventions to simplify the build and deployment process. 21

MIFARE MIFARE are integrated circuit chips used in contactless smart cards and proximity cards. This technology is based on the ISO/IEC 14443 Type A 13.56 MHz contactless smart card standard. It uses AES encryption standards. 40

REST REST (Representational State Transfer) API, or simply RESTful API, is a set of architectural principles and guidelines for designing networked applications. It is a widely used approach for creating web services that allow different systems to communicate with each other over the internet. REST APIs are built on the HTTP protocol and utilize standard HTTP methods (such as GET, POST, PUT, DELETE) to perform operations on resources. 6

SOAP SOAP is a protocol for exchanging structured information in web services over a network. SOAP is based on XML (Extensible Markup Language) and provides a standardized way for different systems to communicate with each other. 20

Spring boot Spring Boot is an open-source framework built on top of the popular Java framework, Spring. It provides an opinionated approach to building Java applications, with the goal of simplifying and accelerating the development process. Spring Boot is designed to be easy to use and configure, allowing developers to quickly set up and deploy production-ready applications. 20, 22

TCP TCP stands for Transmission Control Protocol. It is a core protocol of the Internet Protocol Suite and is widely used for reliable and ordered transmission of data over computer networks. A reliable and persistent connection between the sender and the receiver before transferring data. This connection is established through a three-way handshake process, where the sender and receiver exchange control packets to synchronize and agree on the parameters of the connection. 17

UDP UDP stands for User Datagram Protocol. It is a communications protocol that is part of the Internet Protocol Suite, commonly referred to as TCP/IP. UDP is a connectionless protocol, which means it does not establish a persistent connection between the sender and the receiver before sending data. 17

Unix time Unix time is a system for representing points in time as a single number, specifically the number of seconds that have elapsed since midnight UTC on January 1, 1970 (not counting leap seconds). 25, 36

UUID UUID stands for Universally Unique Identifier. BLE uses UUIDs to identify different attributes, services, and characteristics exchanged between devices. A BLE UUID is a 128-bit number represented as a sequence of hexadecimal characters. 29

Vert.x Vert.x is an open-source, reactive, and polyglot toolkit for building high-performance, scalable, and event-driven applications. It provides a lightweight and flexible framework for developing asynchronous and concurrent applications that can handle a large number of connections and requests with minimal resources. 20

XML XML stands for Extensible Markup Language. It is a widely used markup language designed to store and transport structured data. XML is considered a standard for data exchange and storage as it allows the representation of hierarchical, self-descriptive data in a human-readable format. 12, 59