SYSC4907 PROJECT:
SENSOR-BASED ACCESS CONTROL SYSTEM

By
Jessica Morris, Richard Perryman, Craig Shorrocks
April 2017

A Fourth Year Project Report
submitted to the Dept. of Systems & Computer Engineering
in partial fulfillment of the requirements
for the degree of
Bachelors of Engineering

Abstract

This report describes a Sensor-Based Access Control System (SBACS). The main purpose of this system is to digitise physical means of access, such as keys, and storing them in the cloud. This makes the keys more secure, transferable, and convenient. Included in this report are descriptions of various use cases for the SBACS.

The SBACS is composed of three primary components: the locking hardware, the cloud server, and the user application. This report describes how each of these components was designed, implemented, integrated, and tested. The locking hardware system uses a modular framework, allowing new means of authentication to be added to or removed from the lock as desired.

Acknowledgements

While this report outlines the efforts we have made, this project would not have been possible without the kind support and help of many individuals. We would like to extend our sincere thanks to everyone who assisted us with this project.

We are highly indebted to our supervisors, Dr. Shikahresh Majumdar and Dr. Chung-Horng Lung, for their guidance throughout the project. We owe a lot of our success to their support throughout the 2016-2017 academic year. In addition, we would like to thank Jenna McConnell and Madeleine Ibrahim from the Department of Systems and Computer Engineering office, for their assistance with the administrative tasks relating to the project. In particular, we would like to thank them for arranging a desk for us in one of the Real-Time and Distributed Systems labs, so that we could have a space to store the project. We would like to thank Jennifer Wolters, from the Faculty of Engineering and Design, for allotting us a grant of $160 from the Capstone Design Project Fund. We extend our appreciation to Nagui Mikhail, who lent us his soldering expertise to help prepare some of the hardware components of the project. Finally, we thank Michel and Normand Leroux, for their preparation of the demonstration end unit which was displayed at the Systems and Computer Engineering poster fair.

Contents

List of Figures

List of Tables

List of Abbreviations

AID	Application Identifier
APDU	Application Protocol Data Unit
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWS	Amazon Web Services
COBS	Consistent Overhead Byte Stuffing
CPU	Central Processing Unit
DC	Direct Current
EM	Electromagnetic
FSM	Finite State Machine
GPIO	General-Purpose Input/Output
HMAC	Keyed-Hash Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IO	Input/Output
IP	Internet Protocol
IRQ	Interrupt Request
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LAN	Local Area Network

LED	Light-Emitting Diode
MISO	Master In, Slave Out
MOSI	Master Out, Slave In
MVC	Model-View-Controller
NFC	Near Field Communication
NPM	Node Package Manager
OS	Operating System
PIN	Personal Identification Number
RAM	Random-Access Memory
REST	Representational State Transfer
RFC	Request for Comments
RTP	Real-time Transport Protocol
RTSP	Real Time Streaming Protocol
SBACS	Sensor-Based Access Control System
SBC	Single-Board Computer
SBM	Single-Board Microcontroller
SCK	Serial Clock
SHA	Secure Hash Algorithms
SPA	Single Page Application
SPI	Serial Peripheral Interface
SSEL	Slave Select
SSL	Secure Socket Layer
TFTP	Trivial File Transfer Protocol
UART	Universal Asynchronous Receiver/Transfer
URL	Uniform Resource Locator
UML	Unified Modeling Language
USB	Universal Serial Bus
USD	United States Dollars
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Project Motivation

As technology becomes more and more prevalent in our lives, our identities become increasingly intertwined with the technologies that we use daily. This melding has become extremely prevalent in some areas. Some example areas are professional networking with LinkedIn, and banking with the advent of online account management. Over half of all smartphone users have used mobile banking [1]. This popularity may be derived from how convenient and secure the online handling of money is. However, certain aspects of our day to day lives have not yet been graced by the benefits of electronic security and automation.

Physical locks and keys are still widely used for several tasks where electronic locks could be used instead. House locks, bicycle locks, and locker locks are frequently physical locks. Changing to electronic locks could help streamline all of these locks, reducing the number of keys to remember and improving security by reducing the likelihood that the key could be faked by an attacker. This would also reduce the risk of misplacing keys, as a digital key can be backed up for use on another device. It can also increase the security of the lockers by reducing the number of points of failure. Some applications for this have already been found: for example, Walmart has a system where customers can order products to be placed in lockers with electronic keys called Grab-and-Go [2]. Such a system greatly reduces the work involved in

getting a key to the customer, as a PIN can be sent and received electronically.

1.2 Proposed Solution

This project outlines a Sensor-Based Access Control System (SBACS), that will expand upon the concept of using electronic locks to further tie security and identity to the electronics we use most: our phones. Using near-field communication (NFC) sensors, along with other electronic authentication methods, the identity associated with a phone can be used as identification for anything requiring access control. This represents a huge advantage with respect to convenience, and it would even further lower the number of possible failure points in security. In addition, with modern mobile devices being automatically backed up to cloud storage, recovering access control keys when a device is lost or stolen can be accomplished easily and securely.

While the focus of the implementation will be on utilising a handheld mobile phone as the vector for transmitting keys to locks, the SBACS is designed to be easily extensible to other hardware. This could include devices as simple as a PIN keypad, or as complex as facial recognition. This allows the SBACS to adapt to particular use cases more effectively by accommodating security, user experience, and redundancy needs dynamically.

1.3 Accomplishments

The final product of this project accomplishes the main goals set out in the proposal: a system for access control, using multiple sensors for different methods of authentication, with the ability to dynamically grant and remove access to different users. We have a working prototype that is a drawer secured with an electronic lock, connected to our system to control access. This prototype is able to function with multiple methods of authentication: a PIN and an NFC sensor. An Android application has been developed to allow users to manage aspects of their identity and view information about which locks they are able to access, as well as to use NFC to open a lock. A cloud server stores information about which users are able to access which locks, and

verifies that given authentication information is adequate to provide access to a lock. The lock hardware and the Android application communicate with the cloud server, and the Android application also sends information to the lock hardware using NFC. Finally, there is a web interface to allow management of access to different locks, and to provide basic account management to users of the system. For the simplicity of the prototype, the distinction between administrative users and regular users does not exist, and users cannot be associated into groups like companies.

1.4 Overview of Report

The remainder of this report will detail the work that was performed to complete the work on this project. Chapter 2 will outline the details of how the project was performed, including the health and safety factors that were considered, the professionalism in the project, the management of the project, and the contributions of each member of the project. Chapter 3 will provide some extra information to provide a better understanding of the terminology of the report and the technologies that were used in this project. Chapter 4 will provide some possible applications of the system, and provide an explanation of why these solutions would be beneficial over existing solutions. Chapter 5 will discuss the technical and design decisions to accomplish the goals of the project. Chapter 6 will provide an overview of some implementation details of the project, that were discovered beyond the design phase of the project. Chapter 7 provides details on testing of the system, as well as some of the larger issues that were discovered and fixed while developing the system. Chapter 8 will provide a summary of the report and the work that was performed on the project. It will also detail any areas for improvement or extension of the project.

Chapter 2

The Engineering Project

2.1 Health and Safety

The main health and safety concern for this project was ensuring the safe preparation of circuits for the hardware component. The change of bodily harm due to electric shock was possible, as the highest-risk component of the project drew 650 mA of current at 12V direct current. Since 650 mA is above the 500 mA threshold for possible heart fibrillation [3], the following standards were enforced to mitigate safety risks when handling the electrical components:

- The workspace and components were kept dry at all times
- Participants ensured they were sufficiently grounded before handling components
- Ground pins were connected first, to ensure any charged components would discharge
- Components' wiring was not altered while the system was powered on

2.2 Engineering Professionalism

Engineering is a profession, so professionalism must be maintained when performing engineering projects. In our project we have ensured professionalism in a variety of ways. A professional relationship between the members of the team has been established, ensuring that all members of the team are respected and considered as equals. When disagreements arose about decisions for the project, they were discussed as a team and all viewpoints were understood before making any final decision. This was done to ensure that all members of the team were satisfied with the direction of the project, to ensure the best possible outcome.

In this project security is a very important factor, therefore all necessary steps were taken to ensure that personal information and other secure data was handled correctly. This system may also be used to secure physical objects, so the necessary steps were taken to ensure that only the allowed individuals would be able to gain access to private information, or objects.

In order to understand all background information to construct the system, we used external resources to expand our understanding of certain subjects. We made sure to reference the works of others and acknowledge the help that we have received, in order to give credit where it is due.

2.3 Project Management

Due to the project's complexity, we decided that project management was a necessary tool to facilitate the success of the project. We used GitHub to manage various issues and tasks, as well as for a central version-controlled repository [4]. We held weekly meetings to follow the Agile development methodology, doing one-week sprints. In these meetings, we demonstrated to each other and our supervisors our progress during the sprint. In our project proposal, we planned a timeline that we used to gauge our overall progress. When we fell behind on our planned timeline, we isolated areas of work that needed to be accelerated to get back on track. Responsibilities for the project were divided evenly among the three team members to balance the

work load, and effectively make use of each member's time and specialisations. To facilitate system integration, we held informal meetings where we collaborated on our work. Figure 1 is the project timeline, represented as a Gantt chart. We used this chart to monitor our progress throughout the project's lifecycle.

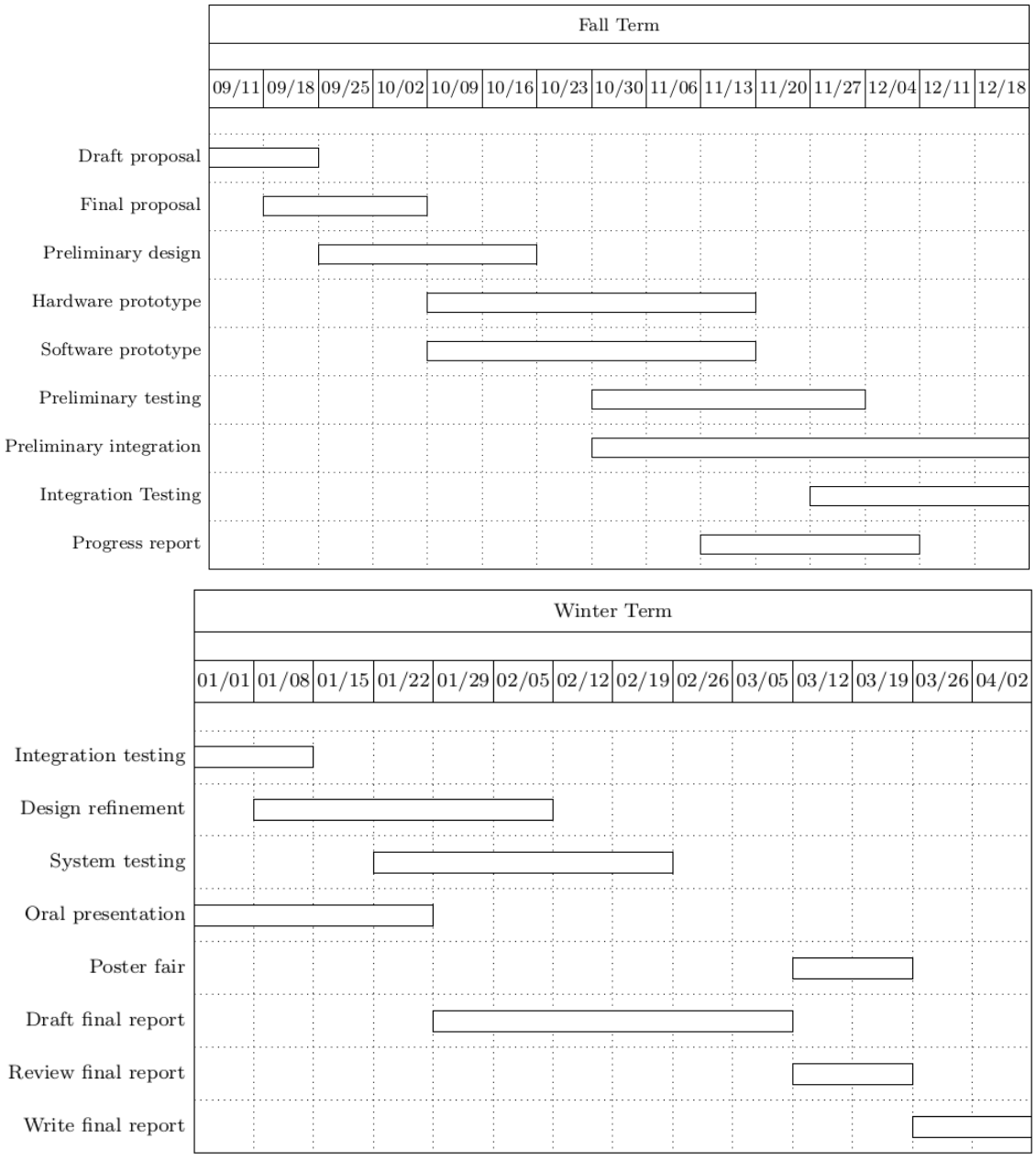

Figure 1: Gantt charts for project management

2.4 Individual Contributions

This section will summarise the contributions each team member made to the project, and to this report.

2.4.1 Project Contributions

Jessica's contributions:

- Designed lock circuit, lock control software, and authentication modules
- Implemented lock circuit, lock control software, and authentication modules
- Collaborated on the design of the NFC protocol

Richard's contributions:

- Designed the Android application
- Implemented Android application
- Collaborated on the design of the NFC protocol
- Collaborated on HMAC design and implementation

Craig's contributions:

- Designed web server and database
- Implemented web server and database
- Collaborated on HMAC design and implementation

As a group, we collaborated on system integration, and integration testing.

2.4.2 Report Contributions

Jessica's contributions:

- Section 3.1, 3.4
- Section 5.4, 5.6
- Section 6.2
- Section 7.1.3
- Section 7.2.1
- Appendix A

Richard's contributions:

- Section 3.3
- Section 5.1, 5.2, 5.3
- Section 6.1
- Section 7.1.2
- Section 7.2.3

Craig's contributions:

- Section 3.2
- Section 5.1.1, 5.5
- Section 6.3
- Section 7.1.3, 7.2.3

All members collaborated on sections 1, 2, 4, and 8.

Chapter 3

Technical Background

This section identifies several technical topics that are relevant to the SBACS. These topics are discussed lightly here to ensure the reader's familiarity with them in later sections. Therefore, these sections do not discuss the SBACS in detail, but instead attempt to set up a baseline from which the SBACS's operation can be understood.

3.1 NFC

NFC is a form of short-range, low-power communication used by devices such as smartphones and tablets. NFC is a fast and convenient method to exchange small amounts of data, as it does not require any steps to set up a connection. The active device uses magnetic induction to create a current in the information-holding passive device. The passive device responds by modulating the EM field coming from the active device. The active device then reads and converts the modulations into useful data [5]. This scenario is a NFC communication in passive mode. Two smartphones may both act as passive and active devices, allowing them to exchange data through a call-and-response procedure, also known as communication in active mode.

NFC is being increasingly used to add smart features to passive information delivery systems such as business cards, and posters. Brief data such as contact information, URLs, or credentials may be written to a passive NFC device, such as a smart tag [6], and read by any NFC-enabled mobile device. These mobile devices can also be

used to replace credit cards in contactless exchanges. In fact, NFC payments may be more secure than payment with a card, as each point in the transaction requires the device and the reader to exchange an encrypted password, and the transaction must be approved by the device's user before the device sends payment information [7].

Because of the close range required for an exchange, NFC has inherent protections against attackers. An NFC exchange can only be reliably eavesdropped from a distance of approximately 10 metres or less if the interaction is between two active devices, dropping to 1 metre if the interaction is a passive communication [8]. This, along with some other facts, makes a man-in-the-middle attack nearly impossible to accomplish in a real-world scenario [8]. These attacks may be protected against by establishing a secure channel, by using symmetric-key encryption or other secret-sharing method. For these reasons, NFC is a reliable method to pair a smart device with an electronic lock system.

3.2 Cloud Computing

Cloud computing is about performing computation, providing data storage, and offering other services on demand over the internet [9]. For this project, Amazon Web Services (as discussed in Section 5.5.1) has been used as the provider of a cloud computing platform to host the server and database of the SBACS. Cloud computing allows for computational resources to be dynamically allocated based on the demand at the time. This means that when a server has low demand it will not use as many resources, but when the demand is higher, it will dynamically be provided with more resources.

3.3 Security

The three main principles of network security are confidentiality, message integrity, end-point authentication, and operational security [10]. Confidential networks ensure that only the sender and receiver can understand the contents of a message. A message's integrity is protected by a network if the contents of the message cannot

be modified by either a malicious attacker or by accident. Networks enable endpoint authentication by providing mechanisms by which the sender and receiver can confirm the identity of the other party. Operational security deals with the ability for organisations to protect their host machines from attackers on the internet.

Confidentiality is primarily provided by means of cryptography. Encryption describes a process by which a message is modified using some secret information [10]. Then if a sender and a receiver share some secret information, they can be sure that only they can understand the actual message. Usually these secrets are called keys. If the shared keys are identical, it is called symmetric-key cryptography. If only one of the sender or receiver knows both keys, but both know of the other key, then it is called a public key cryptography.

Message integrity can be handled in many ways, but the goal is always to verify the sender's identity, and to verify that the message has not been changed [10]. Two common ways of providing message integrity are through message authentication codes and digital signatures. Message authentication codes are redundant information added to a message after being generated using a shared secret value. The sender and receiver must share the secret in advance and follow the same procedure to calculate the code. Then the receiver can check that their calculated value matches the value sent by the sender. Digital signatures are primarily handled by certification authorities. The authority does work to ensure the identity of a potential sender, and then combines that sender's public key along with identifying information using the authority's private key. This way, a receiver can check with the certification authority that the sender is who they say that they are.

Operational security primarily lies outside the scope of the SBACS and is handled by things like firewalls [10]. Therefore, the SBACS does not make any considerations for handling operational security. This means that the SBACS is as vulnerable to operational attacks as the network that it is being run over. As an example, if an attacker had access to a sufficiently powerful machine, they could perform a kind of replay attack. By creating a copy of the data and simply brute forcing all possible keys, the attacker could gain access to a lock.

3.4 Single-Board Computers and Microcontrollers

A single-board computer, or SBC, incorporates all of the elements necessary for a functional computer on a single circuit board: a microprocessor, memory, IO, and other functions. Single-board microcontrollers (SBMs) have the same intent: to incorporate all of the elements needed to support a microcontroller onto a single circuit board. The main distinction between SBCs and SBMs is the same as that between computers and microcontrollers. A microcontroller is designed specifically for one purpose, such as powering an alarm clock. As such, it does not require a significant amount of memory, processing power, or electricity to perform its task. Whereas a computer is a general-purpose computing device, and has more memory, a stronger CPU, and other improved hardware features to support the more intense computing needs of a user. Therefore, SBCs use more energy and are less specific than SBMs.

3.4.1 Raspberry Pi

The Raspberry Pi is a credit-card-sized, single-board computer developed by the Raspberry Pi Foundation. In its initial design, it was intended to be a learning tool to introduce computer science to students at the pre-university level [11]. However, its reasonable price gave it increased appeal among hobby electronics enthusiasts, and it is now often used for robotics projects, media streaming, IoT projects, and more. Today, the Raspberry Pi has sold over five million units [12], with models ranging from the US$5 Raspberry Pi Zero, to the US$35 Raspberry Pi 3 Model B.

The Raspberry Pi has various features which make it useful for small computing projects [13]. In particular, it has 40 general-purpose IO pins, which can be used to read or write digital values. They are useful for driving digital sensors, such as light sensors, or digital devices, such as LEDs. Additional features of the Raspberry Pi include a camera interface, for connecting a Camera module, and a micro SD card slot, where the operating system and files are stored.

3.4.2 Arduino

The Arduino platform is used for building electronics projects. Arduino consists of the hardware component, a single-board microcontroller, and a Integrated Development Environment for writing Arduino code [14]. The Arduino platform is easy to use, as unlike other microcontrollers, the Arduino SBM only requires a USB cable to program. Some varieties of microcontroller require a separate piece of hardware, called a programmer, to load programs [15].

The Arduino SBM can be used to accomplish a variety of small tasks, from driving a PIN button matrix, to reading temperature sensors, to running motors. For these reasons, the Arduino platform was useful for driving the various sensors used in this project.

Chapter 4

Business Use Cases

Design for our system began with considering use cases that could be handled by our system. This way, we could focus on designing features that would be useful to the majority of targeted users. These use cases would also provide benefit to demonstrating our system, in that stepping through a sample use case would show the value of the system to people without them requiring a strong technical background.

We primarily focused on the four use cases that follow. These were outlined in our initial proposal and have remained largely unchanged since then. They draw inspiration from situations that we either encountered ourselves or found through common knowledge and research. SBACS fit well into the existing online shopping infrastructure, and mail delivery follows a fairly similar system. Long term storage facilities also seemed as though they would benefit digitising keys. Once we had considered these use cases, we found that several of the problems that they shared could be solved by a company which provided SBACS as a service. The following subsections describe these use cases in more detail and provide use-case diagrams summarising them.

4.1 Online Order Secure Pickup

With online shopping becoming more popular [16], the hassle of being physically present to shop in stores could be eliminated by allowing customers to make an

order and payment online, then pick up their products at the store using a SBACS-secured storage container. After a customer's order has been confirmed by the store to be packed and ready for pick-up, a store employee could create a link between their customer and the storage container that holds that user's purchases. When the customer arrives at the store to pick up their items, they can gain access to the storage container assigned to them by using the authentication methods that they had previously provided as the components of their identity. Once the customer has provided these authentication units correctly, the link between their identity and the storage container is broken. This way, the user will not be able to inappropriately access the storage container again, and a new link could be made for a different customer. Figure 2 shows a use-case diagram summarising this.

A strong advantage of this system over existing ones, such as Walmart's Grab & Go, is that it creates a digital fingerprint using a combination of authentication methods supported by the SBACS, which is more difficult to break via brute force than a 6-digit PIN. In addition, with the entire process of shopping being expedited, customers will spend less time in stores and parking lots, leading to less congestion due to vehicle traffic. This system will also grant customers with mobility or vision impairments increased personal autonomy, as they will be able to retrieve their goods with less aid from store employees than traditional shopping allows. Customers with impairments can perform their shopping online using their accessibility software, such as text-to-speech, instead of calling the store to request that an employee retrieve their goods. When the customer receives confirmation of their order and authentication tokens, they can access their assigned SBACS-secured locker in the store themselves, using their usual aids instead of occupying a store employee's time.

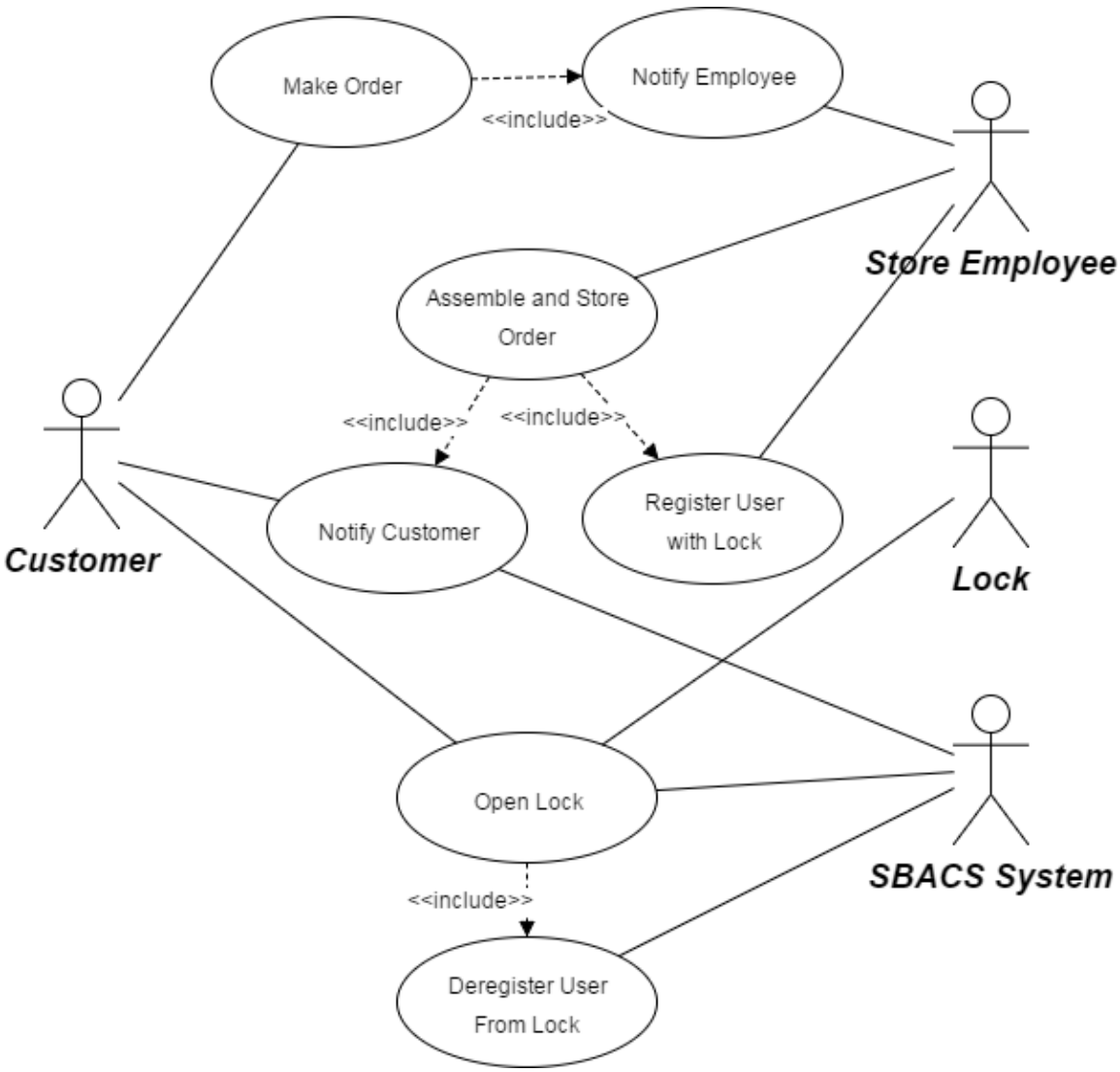

Figure 2: Online order secure pickup use case diagram

4.2 Central Mail Package Pickup

Some parcels are too large to fit in a regular mail box, and must be delivered in person. Many delivery services only operate during standard work hours, while many people that need to accept the delivery will not be home. In these cases where packages cannot be stored in a mail box, and cannot be accepted by the recipient, these packages often go to an office of the delivery service. The recipient must then go to pick up their package at the office. This is counter-productive, as the customer has paid for the delivery, and now they are forced to go out of their way to get their item, which has not been delivered to their desired location.

In some places, Canada Post already has community mailboxes, which have a box for larger parcels. The problem with this is that it is only usable for parcels that are being delivered with Canada Post, but there are other parcel delivery services as well that are unable to make use of this. This community mailbox also uses a physical key to open this box, where they leave the key in the customer's normal mailbox, then they use that key to open the parcel box, then they must return the key by dropping it in a mail delivery slot. This involves the need for the physical keys, and requires staff to physically find the keys for the parcel box among the other mail that is in the delivery slot. The Sensor-Based Access Control System would eliminate the need for these physical keys, and would allow more delivery services to make use of the same boxes, ensuring that it is most convenient for the customers.

This system could be used to access parcel boxes, which would allow the delivery to be placed in a box that would fit the parcel. These boxes could be set up in an area where they are close to many customers, allowing them to quickly pick up their parcel without going all the way to the local office. The delivery person would grant access to the customer, then place the parcel in the box. Then the customer could be notified through the application that there is a parcel ready to be picked up. When the customer arrives at the box, they can use the application to open the parcel box, and receive their package. These interactions are summarised in the use case diagram in Figure 3.

Throughout this process, the system could make use of the registrations that have

been set up for the central mail box. The user set up as the deliverer would be able to send notifications to the system on the status of the package, allowing the recipient user to better know when their package would arrive. Further, the system could be configured such that the system would notify the user when the package had not been picked up after a certain time had elapsed since it was dropped off. This information could also be exposed to the people who manage the central mail box, allowing them to determine the status of their boxes, informing them of potential problems, such as a package that has been languishing for too long in the same box. If the package is not picked up after a given amount of time, the system would notify the customer that the package will no longer be available to be picked up, and alternate options will be given. This would also lead to an expiry of the customer's registration to the box, so they will no longer be able to open the box.

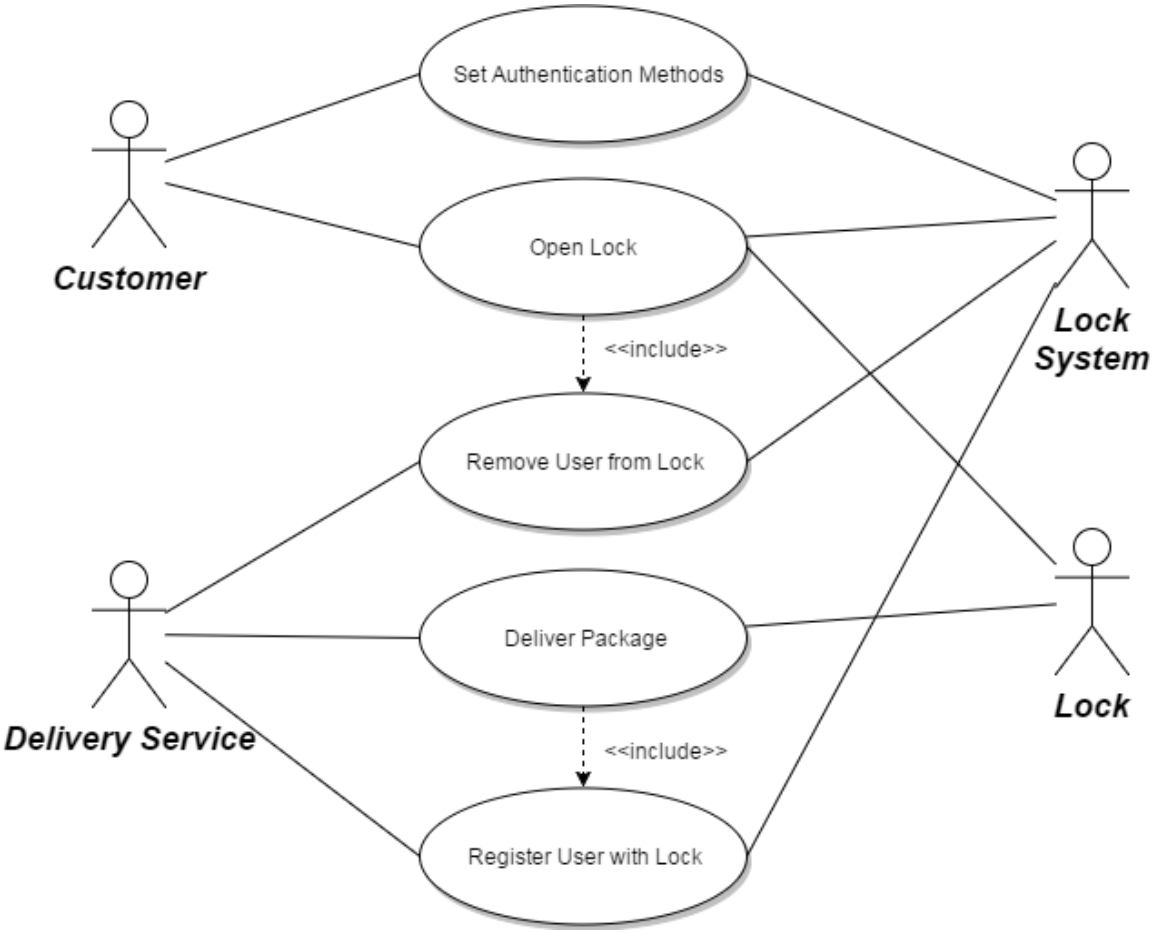

Figure 3: Central mail package pickup use case diagram

4.3 Long Term Storage

The Sensor-Based Access Control System also has applications for long term storage companies, such as Dymon. For these companies, the system could be viewed as an overall manager of customer storage. The system's registrations can be used to provide additional services for specific storage spaces. The system will also expose much of the administrative tasks that the company will need to provide.

The system, using its multiple step authentication, would allow the company to advertise greater security. Along with similar methods to those described in the use cases in sections 4.1, and 4.2, the system could be configured to require biometrics, such as fingerprints or facial recognition, as a form of authentication. By combining these metrics, the system would greatly reduce the potential for attack by strengthening the potential failure points of the identification process.

Once a customer has a storage space reserved and has set up their identification, the registration could be configured to expose additional features. An example feature would be a live video stream of the contents of the storage space. This would allow the customer to verify that the contents of the space had been correctly stored, for example after requesting that the company switch their locker. Figure 4 shows a use case diagram outlining the interactions to manage a long term storage facility with SBACS.

Another additional feature would be the option to log items that are stored in the storage space. Through the system, the customer would be able to note which items they are adding or removing from the storage space, which would later allow the customer to remotely access a list of the items in the storage space, so they do not have to remember all of the contents. This provides a different view of the stored contents on top of the video stream mentioned above. This list could also be searchable or contain images of the items, to make the list easier to use.

The system would also provide the ability to dynamically give certain users access to the storage room. This would allow a storage room to be communally owned by the members of a conglomerate, for example a company. This would ease the process by which multiple people may access the storage space while also ensuring

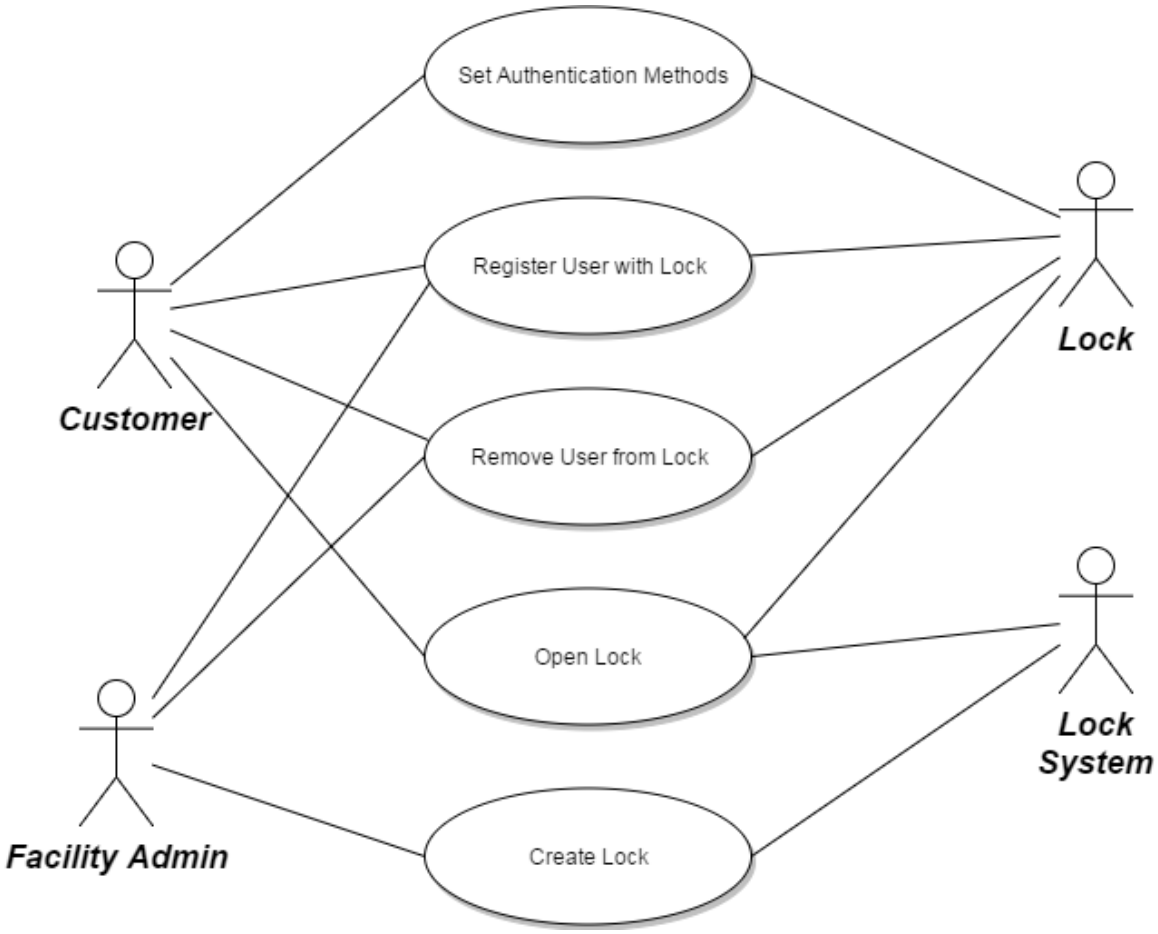

Figure 4: Long term storage use case diagram

that control of the storage was held centrally. More privileged users would be able to allow temporary access to less privileged users for a given time period. All of the attempted authentications could also be logged and made available to certain users, creating a reliable paper trail.

4.4 Service Provider

Since one of the main goals of the Sensor-Based Access Control System is to allow for a user to share their identities between different locks, a logical application of the system is a service provider. A service provider could sell the service of installing and

managing the information technology parts of the system to other companies that might want to make use of it, such as Walmart or Dymon mentioned above.

This would benefit the end users as they would be able to share their identities amongst the service providers' various customers. This would allow someone to sign up for a locker at Walmart, and later reuse their SBACS identity to open their apartment's mail package pickup unit assuming both SBACS systems were managed by the same service provider. This shared identity would allow people to remember fewer passwords and generally have fewer physical keys.

Customers purchasing the services for the SBACS would save any technical work associated with installing the system. The service provider would be able to manage the databases, servers and any customer complaints or feedback. Providers would also take care of sharing end user identities, which would even further increase how much time end users could save, as they would only be required to set up one account with the service provider instead of one account per company. This incentivises companies to make use of a service provider, as it makes the resulting system more attractive to potential customers as well as reducing the overall cost of setting up the system.

Chapter 5

Problem Analysis and System Design

This chapter describes the initial work done to plan out the overall system, before beginning the implementation work. The following sections highlight the main considerations and factors in our initial decisions when designing the system. As with all systems, some of the design was changed during implementation, which is described further in chapter 6.

5.1 Overall System Analysis

The requirements of our system from the use cases described in chapter 4 lead us to a simple outline for the entire system. The main component of the system would be the cloud server, which would store information regarding users, locks, and the relationships between them. The lock hardware would need to be able to securely accept data from the user, and securely send that data to the server for verification. Since handling electronic security can be difficult to do manually, having an application that handles sending and retrieving data would greatly help users interact with the system.

One of the main goals of the system was to improve the reusability of the electronic keys that are created. For this reason, the idea of a key was split into two parts: an

authenticator, and an identity. Users would be able to create many authenticators, which are data like passwords, or PINs. These then could be combined into an identity, which would be used to open a particular lock. This allowed making identities that reuse the same authenticator, and reusing identities for different locks.

Another goal of the system was to provide additional behaviour beyond being able to unlock particular locks. This extra behaviour was captured in the notion of a registration, which ties one lock to one identity. Through the registration, we were able to implement notifications to the user as well as providing a video stream of the locked container's contents.

5.1.1 Architectural Patterns

The architecture of our system is a Centralized Client-Server pattern [17], as can be seen in the model diagram in Figure 5. The client components of the system are the LockHardware, AndroidApplication, and WebInterface. The server components are the SbacsServer and the SbacsDatabase. This is centralized because the data and general functionality of the system are on the central server components. The client components have some specific functionality locally as well, generally limited to basic processing of information supplied by a user, as well as the manipulation of the user interface.

Each component in the system uses a Basic Layer pattern [18] to divide responsibilities of the component. Specifically, there are four main layers used: user interface layer, system interface layer, function layer, and model layer. The user interface layer handles the display to the user and the input from the user. The system interface layer is used to send and receive information between components in the system. The function layer is used to process information in a variety of ways. The model layer is used to store the needed information of the system.

The AndroidApplication component has a user interface layer, to handle the display of information to the user, in the form of a graphical user interface. The user can interact with this interface and enter information. When information needs to be processed, it is sent to the function layer, where basic processing of the information

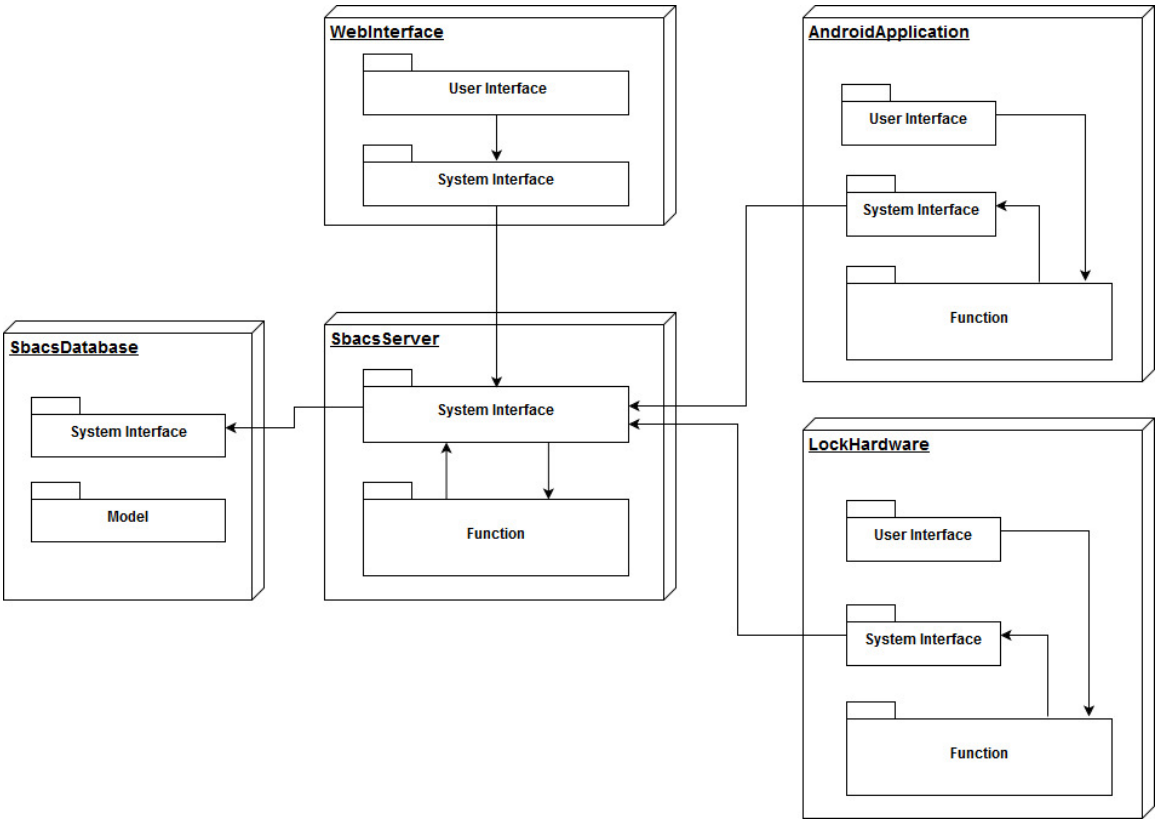

Figure 5: Architecture of SBACS

is performed. In some cases, there will need to be more processing performed, using the functionality available on the central server. In order to send the information to the server, it is passed to the system interface layer to be sent to the SbacsServer.

The WebInterface component has a user interface layer that acts in a very similar way to the AndroidApplication component. However, the WebInterface does not perform any processing of information locally, so it does not need a function layer. Instead, when processing is required, it is sent to the system interface layer to send the information to the SbacsServer.

The LockHardware component has a very basic user interface layer, consisting of LEDs for the display of information and various authentication modules for user input. Once users input information through an authentication module, the information is passed to the function layer to be processed. Similarly to the AndroidApplication, the information sometimes requires more processing on the central server, and in that case is sent to the system interface layer to send the information to the SbacsServer.

The SbacsServer component has no user interface layer, as it should only be interacted with through one of the client components. Instead, all input to the server is done through the system interface layer in the form of HTTP requests. The requests are then passed to the function layer for processing, which generate an appropriate responses. The processing of requests often requires information stored in the SbacsDatabase, so the system interface layer must be called to send a request to the SbacsDatabase.

The SbacsDatabase component only has system interface and model layers, as it is only meant to store information and respond to requests for information. The system interface layer is used to receive requests for data, and then will call the model layer to retrieve the appropriate data. The model layer in the SbacsDatabase is the only model layer that exists in the system, as it stores all of the data for the system as a whole.

5.2 NFC Design

Determining how NFC communication should take place required analysis of two hardware systems: NFC card readers, as well as mobile devices. The most desirable protocol would be able to handle the widest variety of available hardware systems for the two devices. The performance considerations between modes was fairly minimal, so preference was placed on the portability of the solution.

5.2.1 Card Readers

Since NFC cards are primarily designed for NFC communications, there were few restrictions that stemmed from potential choices in card reader. Since NFC communications are specified by the ISO [19], most cards support enough protocols that the decisions on our part would be very likely to be supported by a card that would be desirable for any other reason.

5.2.2 Mobile Devices

The two most popular operating systems for mobile devices are iOS and Android [20]. Since iOS devices have NFC disabled for everything except Apple Pay[21], the only option that remained was Android. Apple devices would represent a large part of the potential market, so alternatives to NFC would have to be considered.

Among Android devices, there exist devices which have hardware support for NFC communications, and those which rely on host-based card emulation. Devices with hardware support have a component called a secure element which performs all of the communication with the external NFC terminal. Later, applications can query this element to determine the status of the transaction, as well as other data. Devices which use host-based card emulation use a software implementation of secure elements. Since host-based card emulation is done through software, it will run on Android devices running version 4.4 or greater [22], which represents over 99% of all Android devices currently in use. The differences between secure elements and emulators can be seen in figures 6 and 7.

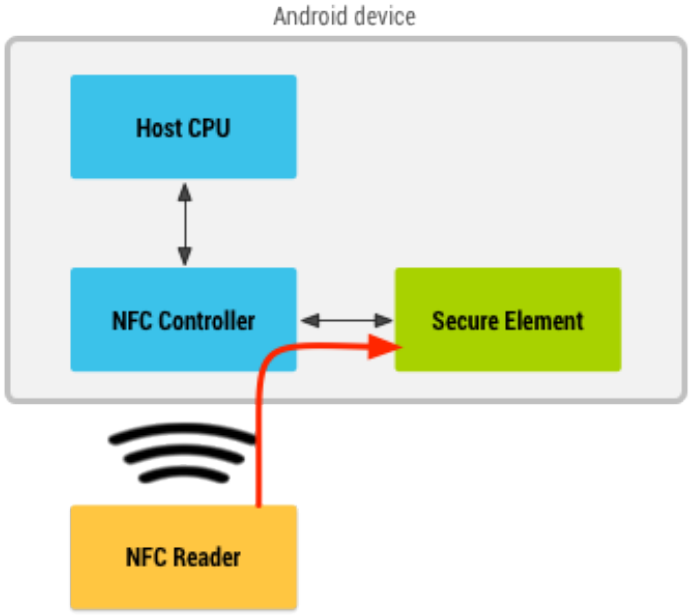

Figure 6: An android device using a secure element [22]

Android offers an API called Beam which is the only way Android devices can use NFC in active mode [23]. Beam, however, does not support sending more than one message between devices. Since the information we are sending can be fairly large in the interest of security, this was not feasible given the restrictions of the NFC protocols we used. Further, active communications are easier to eavesdrop on, as discussed in the background section. We decided that these costs outweighed the simplicity of the Beam API, so passive communications were chosen for the implementation.

Since more than one application could potentially want to handle an NFC message, the message protocol contains a field called the Application Identifier, or AID [24]. These AIDs are 16-byte bit patterns which identify which application should handle the associated message. The Android operating system is responsible for delivering the APDUs to the appropriate application's service [25]. Reserving a particular AID for national or international use requires some paperwork and a fee, as per the international standards organisation [24]. Therefore, by not paying this fee, our application may happen to share an AID with another application. We found that the cost of a potential collision was less than the cost of reserving an AID, therefore our

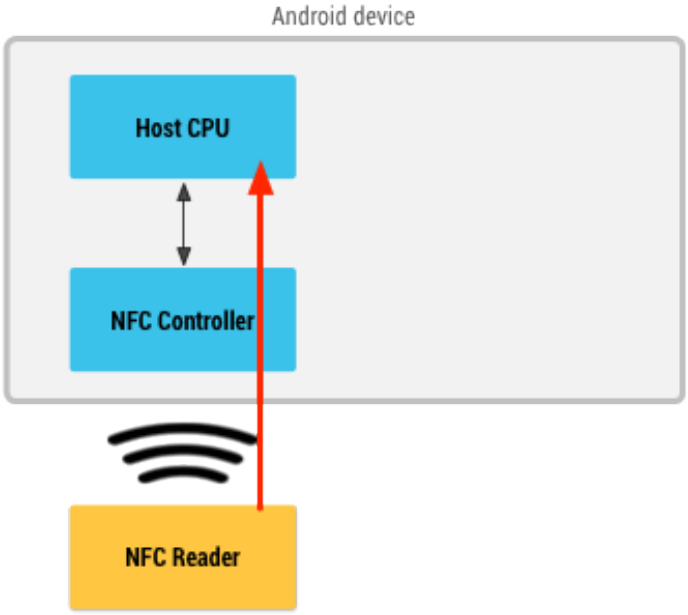

Figure 7: An android device using host-based emulation [22]

application uses a proprietary AID which we selected at random.

5.2.3 Protocol

Since our NFC communications may require more data than can be fit within an Application Protocol Data Unit (APDU), we required a protocol which would handle segmenting and recombining the message. APDUs are defined in ISO 7816-4 [19] and are the units used by ISO 14443-4 [26], which describes the transmission protocol used by NFC devices. They are restricted to 256 bytes, including headers.

To work around this, the hardware connected to the NFC shield maintains a buffer. Under ISO 14443-4, messages can be reliably transferred, so managing this buffer is the main consideration of our protocol. The hardware first fills in the mandatory fields, like the AID. The hardware computes the space remaining in the APDU for data, and adds that value as the length field of the APDU, which it then sends to the Android device. Then, the Android application responds with the minimum of that much data, and all of the remaining data that it has to send. Once the hardware receives an amount of data less than the potential maximum, it deactivates the connection. In the event that the data from the Android application fits exactly into the last message that would be sent, the protocol still works, as the application will then respond with zero data bytes. This process is visualised as a message sequence chart in figure 8, as well as state machines in figures 9 and 10.

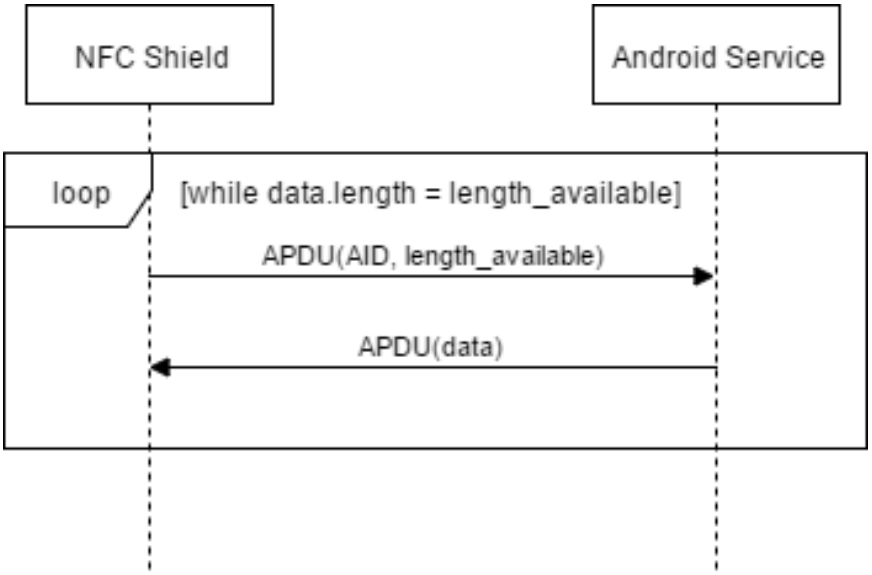

Figure 8: The happy-path message sequence chart for the designed protocol

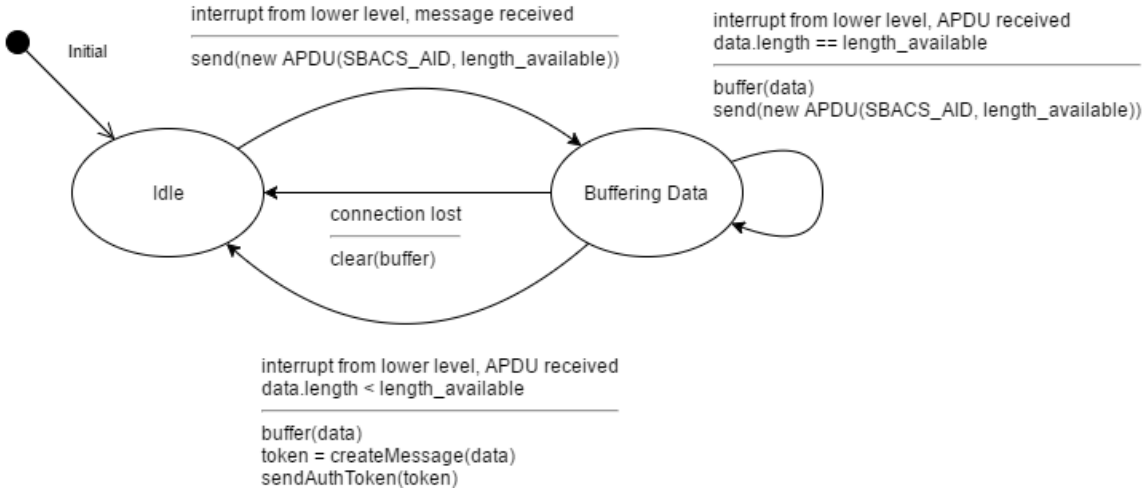

Figure 9: FSM describing the hardware's implementation of the protocol

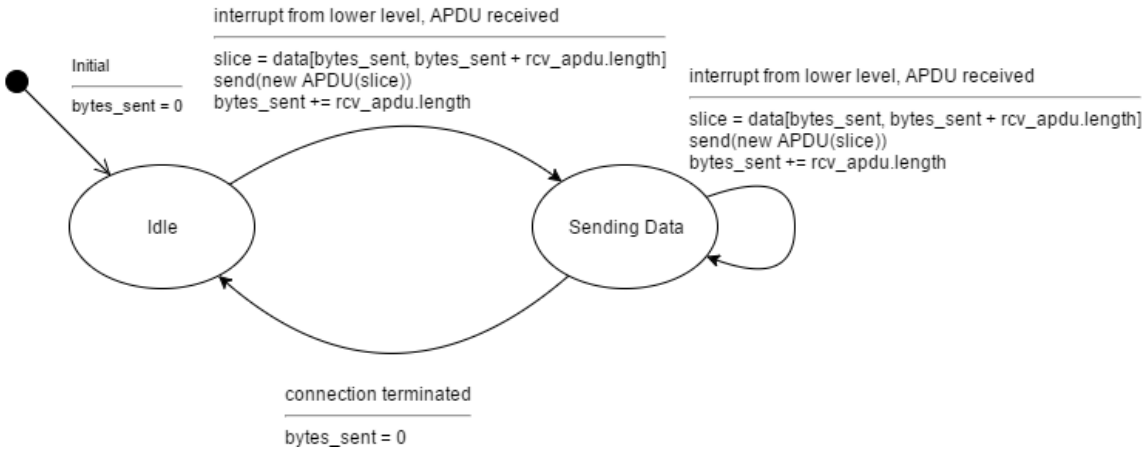

Figure 10: FSM describing the application's implementation of the protocol

5.3 Android Application

The Android application's goal was to provide the end users with an easy way of interacting with the SBACS system. Therefore the application was designed to be as simple as possible while still maintaining flexibility when dealing with the cloud server as well as the many potential hardware devices.

5.3.1 User Interface

The natural decision for designing the basic workings of the application was to use the Model-View-Controller (MVC) [27] architectural pattern. This pattern separates the underlying data (model) from the display that the user sees (view) as well as the components that the user interacts with (controller). Android provides APIs to support MVC. Model objects can be simple objects, but view objects can subclass the View class or its more specific subclasses, and the controller objects can subclass the Adapter class or its subclasses. More precisely, since the View objects and the Model objects never communicate, this follows the Model-View-Adapter pattern. The difference can be seen in figures 11 and 12.

Android applications themselves should follow the patterns set out by the standards [28]. Activity classes represent the pages that contain the various views and controllers that users can interact with. We decided on a design where the user first

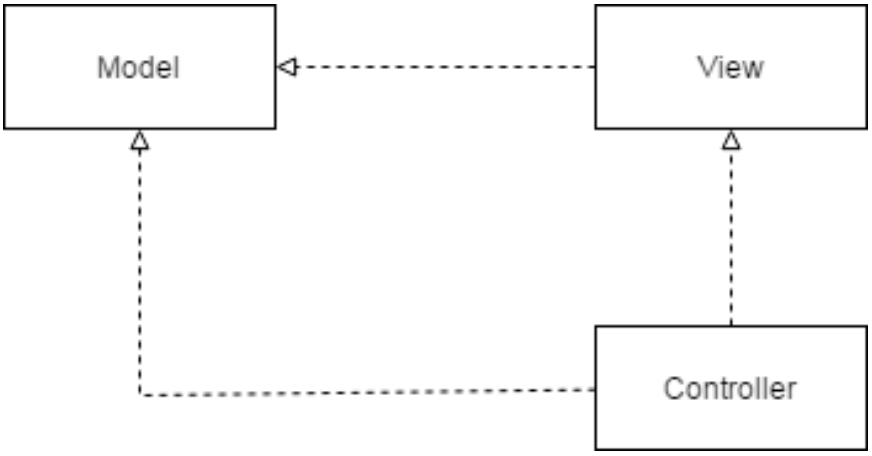

Figure 11: Demonstrative class diagram of the MVC pattern

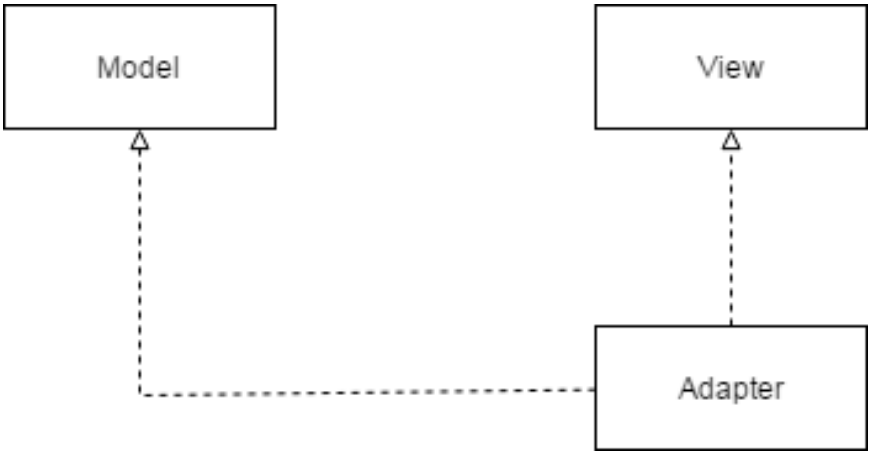

Figure 12: Demonstrative class diagram of the MVA pattern

encounters a login Activity, which prompts them to either sign up or log in. Once the user has logged in, we provide a hub Activity which leads to the various other Activities in the application. These other Activities show the various data related to the user, such as their authenticators and identities.

Communication between Activities is handled by a class provided by Android called an Intent [29]. Intent objects contain information about the nature of the request to begin the new Activity. In this way, information such as the user's identification number could be sent from the login Activity to the hub Activity, which would allow the application to correctly load the information from that user when displaying the data Activities.

5.3.2 Notifications

The design for the notifications that would inform end users of information such as their newly available registration with a particular lock was based partly on decreasing the load on the server. If we had designed the notification system to be handled by the server, we would have needed the server to maintain a large amount of state information. In order to be able to send the notifications, the server would have to remember the IP address of each current user. This would require handshaking between the application and server on creating and terminating a connection, as well as when the application changes IP address due to roaming. Further, the server would have to queue up all of the notification information that a user who does not currently have a session would want. Having the server maintain all of this information was thought to be too great a cost. For this reason, it seemed simpler and more effective to have the application poll the server at a particular access point for new information.

This polling was implemented simply in the application using a service which runs in the background of the Android device. This service regularly hits the server at a particular endpoint designed for handling these notifications. Since many users may be using the application at once, it was important to consider the performance of the endpoint's code. The endpoint returns information valuable to the application for display in the notification. The notifications make use of an Android notification's

ability to launch an Activity with an Intent to take the user to the appropriate Activity associated with the notification.

The overall design of the polling sequence can be seen in figure 13. Some simplifications have been made to the diagram to preserve its readability. The main simplification is the Notification that is created by the service. In order to preserve security as well as to improve the user's experience there is only ever one SBACS notification in existence at a time on a given device. This way when a user logs out, another user cannot see their notifications. In addition, the user will not be inundated with a large number of notifications when, for example, they join a company which makes extensive use of SBACS locks. Therefore, when the service receives new notifications, in the opt block of figure 13, it first checks if there is an existing Notification. If there is, instead of creating a new Notification it updates the current one. Also, Notifications can be cleared by the user visiting the appropriate page of the application that would show the new information, or by the user interacting with the notification through their Android device's operating system.

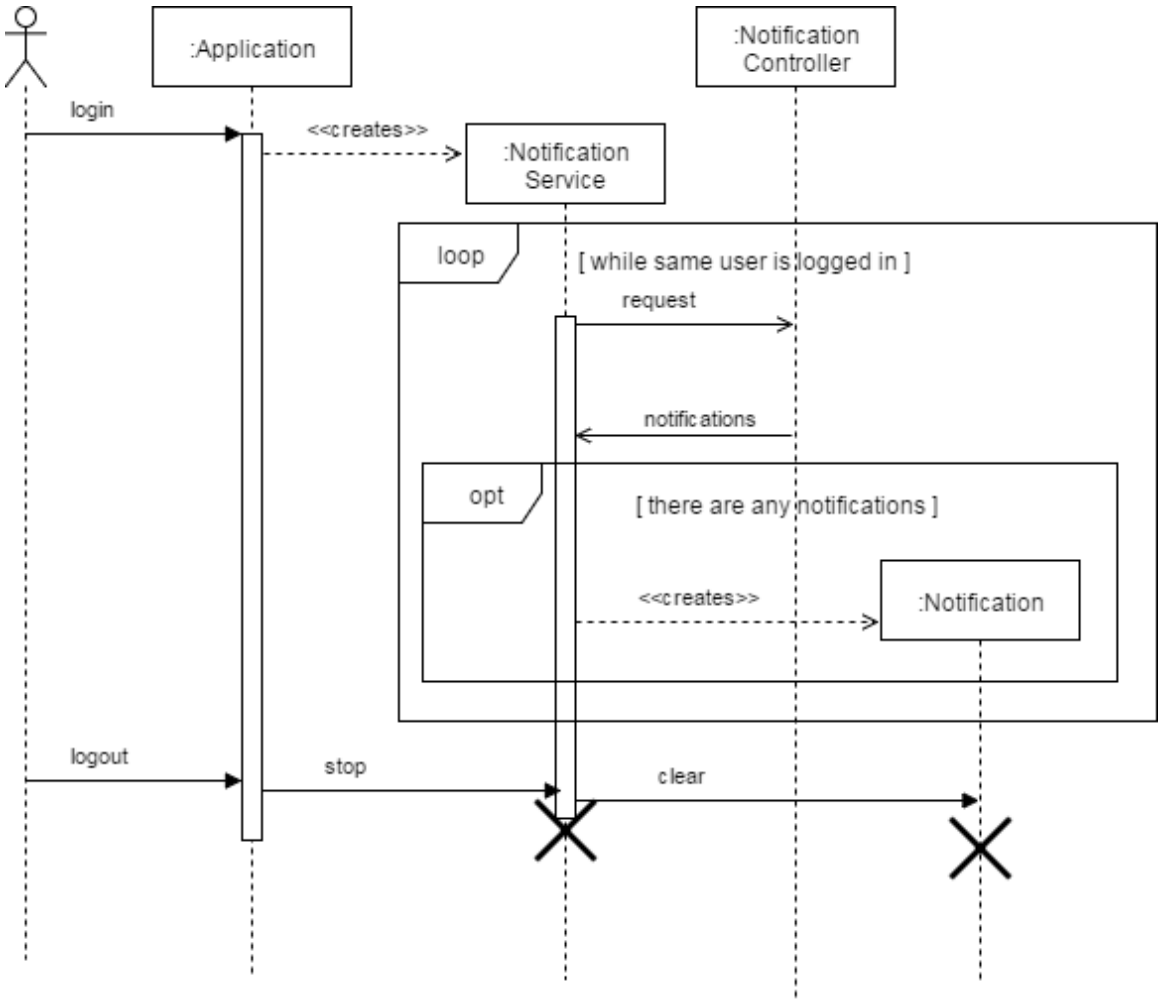

Figure 13: Notification polling sequence diagram

5.3.3 NFC Handler

Recall from section 5.2.2, the NFC communication method that we decided on used host-based card emulation. Android provides an API to accomplish these communications, through the HostApduService class [25]. This class provides a framework for creating a service which runs in the background on the Android device. The service handles setting up and closing NFC communications, while also providing overrideable methods to determine what data should be sent based on the incoming message.

Services also take an Intent object to determine their purpose [29]. This feature allows the login Activity to start the service with the correct key information for a user's NFC authenticator. This way, the data sent by the service could contain the user's secret key value. Since passive NFC communications are difficult to eavesdrop on, this was thought to be a safe method of conveying the secret key to the lock without requiring too much data to be sent over the slow NFC physical links. This passing of information of course was also configured to follow the protocol designed above for NFC communications in section 5.2.3.

5.4 Lock Hardware

The locking hardware consists of three major components: the lock control circuit, the lock control software, and the authentication modules. The control of the lock is kept separate from the access components, as one of the requirements of the SBACS is to provide modular solutions to different access control requirements.

5.4.1 Lock Control Circuit Design

Design of the lock control circuit, the core of the system, began by determining the requirements:

- There should be LEDs to indicate the status of the system, and to notify the user of the results of their access attempt;

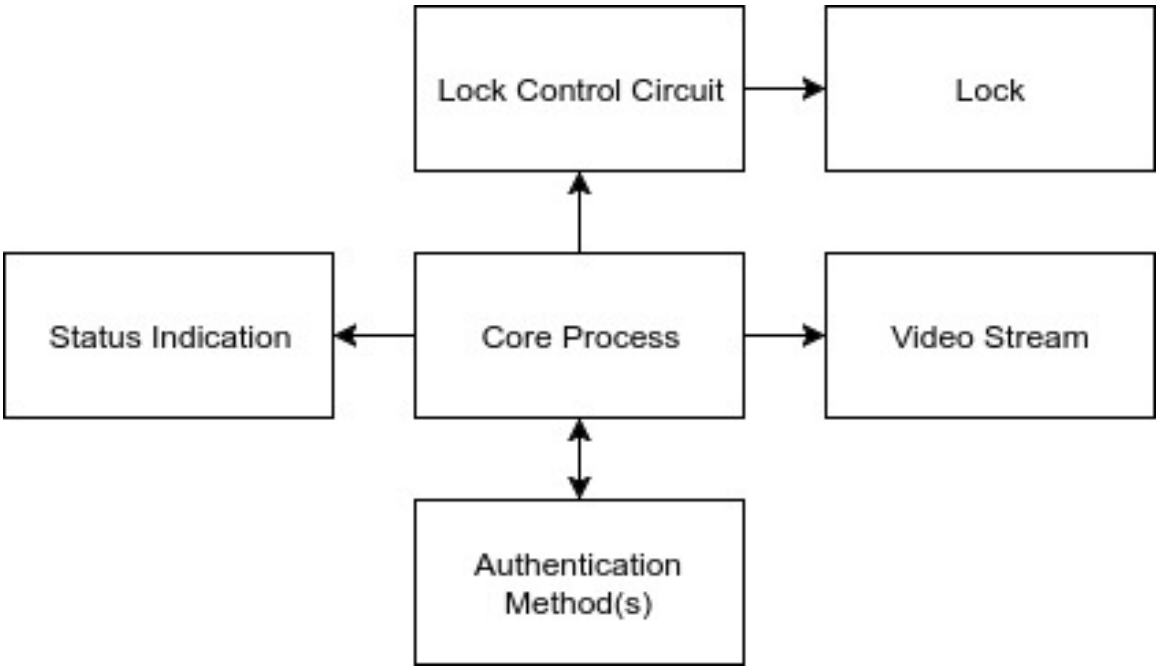

Figure 14: Hardware block diagram

- There must be an electrically-powered lock, and a means to digitally alter its status;
- The core must run a process where all authentication data will be passed, and this process must communicate to the server over the internet;
- The core must have USB ports, to support having modules send authentication data to the core.

Figure 14 is a block diagram of the first draft of the core layout. Note that the specific electrical elements used in each block are discussed in section 6.2.1.

5.4.2 Lock Control Software Design

As mentioned in the requirements stated in the previous section, the lock control software should manage the status LEDs, the lock, and handle incoming authentication data and server interactions. Since the first three elements listed are hardware-related

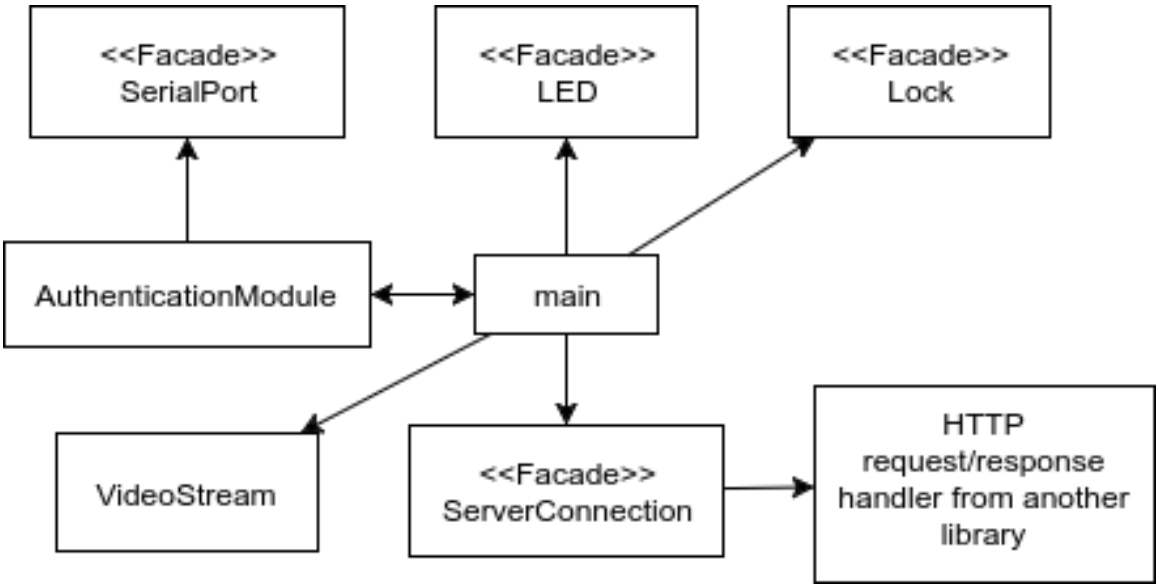

Figure 15: Lock control software block diagram

operations, creating a layer of abstraction between the between the hardware and the core process would increase the cohesion of the design, and prove simpler to implement. This resulted in the first three possible classes for the lock control software: one for the abstraction of the LEDs, one for the abstraction of the lock control circuit, and one for the abstraction of the hardware interface used for communicating with the authentication modules. Since USB is widely used and simple to implement, USB was chosen as the means to connect the modules.

Continuing on with dividing the various responsibilities of the lock control software into subclasses, two more emerge from the remaining requirements: one class to poll the authentication modules, which will take advantage of the hardware abstraction of the hardware interface, and one class to handle server communication. Figure 15 is a block diagram of the initial design architecture of the lock software, including the potential design patterns that the classes should use. The hardware abstraction classes and the server communication class are Facades, as they make interfacing with the hardware and server easier for the main class [30].

The primary goal of the design is to render the central logic, contained in the main class of the program, as simple as possible. By abstracting the various components

of the systems to adapters and facades, the main class would not need to be aware of the specific hardware implementations of the LEDs, or the authentication modules. In addition to its simplicity, this design is also easier to test, as unit tests could be written for the classes that do not rely on hardware-fired events.

5.4.3 Authentication Modules

Authentication modules for the SBACS can come in many forms, so the software for the modules needed to be written with a general structure. Multiple design patterns emerged immediately in the design of the software for the modules. Interfacing with the system core would be the same across all authentication modules, so only the method used to gather authentication data is unique to each module. Two design patterns can be extracted from this statement: an Adapter pattern, where the Adaptee provides the interfaces necessary for communicating with the core [30], and a Template method for handling the authentication data. The Adapter can simply implement the Template method, as well as add any other functionality necessary for the means of authentication. Figure 16 is a basic class diagram of how these patterns would be implemented. The AuthenticationModule class is the Adaptee. Adapters like the SpecificModule class implement the getData() method to complete the sendData() template method.

With an Adapter pattern in place to handle the creation of various types of authentication modules, there also needed to be a pattern to handle the general flow of data, which starts with user entry, and ends with transmission to the system core. The Entity-Controller-Boundary pattern was a natural fit for this situation, and as shown in Figure 17, the first draft of the authentication module software only required one of each stereotype in the pattern, as the software did not need to be complex. An actor, such as a user attempting to gain access, interacts with the Boundary class of the authentication module, the Module-Specific Data Handler, to send a form of authentication data to the system. The Data Handler hands the data to the Module Controller, which then converts the data into an Entity class, represented as the Authenticator class in Figure 17. Authenticator is an abstraction of the authentication

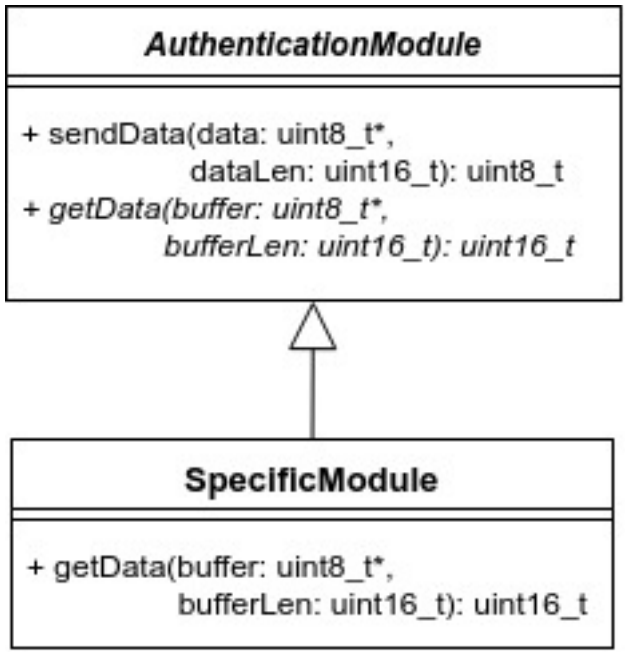

Figure 16: Authentication modules basic classes

data, which would be sent to the core process.

In the planning stages of the project, it was intended that the authentication modules would communicate with a central single-board microcontroller, such as a Arduino, which would be deemed a "communication board." This communication board would be responsible for receiving data from the authentication modules, and for error detection and error correction, removing that overhead from the central computer. Once all of the authentication data was gathered, the board would send the data over a USB connection to the central computer. However, this design was

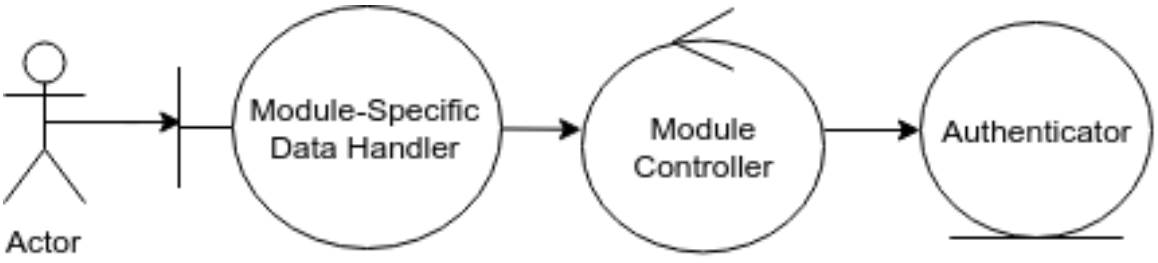

Figure 17: Authentication modules Entity-Controller-Boundary diagram

discarded for multiple reasons:

1. The Arduino, as well as most other microcontrollers in the hobby market, can only communicate with another Arduino over I2C;

2. There is only one pair of pins reserved for I2C on most microcontrollers, meaning there can only be one "slave" for each "master" communication board [31];

3. Connecting all of the authentication modules over USB allowed for far more modularity than the "communication board" would have.

Thus, the design changed, in particular to accommodate the final point. Authentication modules would send authenticators directly to the core system over USB, with no middle board required. This simplified the design considerably, as the behaviour of the authentication modules did not need to change at all, and the responsibilities of the communication board were offloaded to the system core. Overall, this design would require fewer microcontrollers, while still meeting the requirements.

5.5 Cloud

5.5.1 Amazon Web Services

Amazon Web Services (AWS) provides a platform to host various cloud services, including web servers and data storage options [32]. In this project, Amazon Web Services was chosen to host the web server and the database. This decision was made because the platform is versatile and easy to use, providing a great level of control where needed but also handling the complicated configuration details automatically. AWS also provides these services for free within a trial period of one year, making it an even more attractive choice for a project with limited lifetime. To host the web server, the choice of programming language and technology, as well as the choice of operating system to run the server, are left to the user. Similarly, the database technology is a choice of the user, making AWS a very flexible platform to work with. Amazon Web Services also provides some other very helpful features, such as domain

name services, SSL certificate management, and various security configuration details to protect the web server and our data. We chose to use AWS because of the variety of services that were offered that provide everything we needed to effectively host our web server for this project.

5.5.2 Representational State Transfer

The Representational State Transfer (REST) architectural style [33] was chosen to be used for our web server. The main principle of REST that we adhered to is the idea of a stateless server in the client-server relationship. This means that the server does not store any session information; enough session information must be passed from the client on each request to identify the necessary information for the request. The benefit of this for the server is that less information is stored on the server, so less time will be spent contacting the database to determine session information. Another feature of REST is the idea of a uniform interface. This helps to keep the system simple by relying on identifiers to specify which resource or service is needed at the time. The path and the query string in the URL are examples of identifiers that specify which endpoint or service is being requested. The web server for this project has this uniform interface as all HTTP requests go to a single Front Controller (Section 5.5.13), which then determines based on the identifiers the appropriate handler of the request.

5.5.3 JavaScript Object Notation

In order to send information between the client and the server, a standard format for representing the data is necessary. We have chosen to use JavaScript Object Notation (JSON) to serve this purpose. "JSON is a lightweight, text-based, language-independent data interchange format" [34]. JSON provides an easy way of transferring data that is flexible and has support from many programming languages, making it a very good choice for our project, where different components will be using different technologies and built in different languages. It is also very lightweight in terms of its grammar, having very few elements that allow for many different types of data,

keeping the parsing easy which can help to improve response time. Figure 18 shows the JSON representation of Authenticators for a user. This information is a list of Authenticator entries, each with an Auth_Id, AuthType, AuthKey, and AuthSalt, where the AuthKey and AuthSalt each have an array of byte data and type of Buffer.

Figure 18: JSON Representation of Authenticator data for a user

5.5.4 Node.js

Amazon Web Services allows for web servers written in many languages, leaving the choice of language to the user (as discussed in section 5.5.1). In our case we decided to use Node.js to write our server code. This language was chosen because it is able to run on multiple operating systems and does not require more than one tool for effective development. The ability to run on multiple operating systems is very useful to test our code in different ways, and leaves us with flexibility of which operating system to host the server with. In the early stages of the project, we considered using C# with the .NET framework to build the web server, as some members of the group had experience working with this particular technology in the past for a similar purpose. After some discussion, it was decided that in order to effectively develop the application in C# tools would be needed that are not available on the computers at school, which would limit our ability to work on the server. This is a major reason that we decided on Node.js, since it can be very effectively developed using only simple text editors, then can be executed quickly, without needing to install any build tools. This portability made it very easy to work on the code in the same development environment whether we were at school or at home.

5.5.5 HTML

Hyper Text Markup Language (HTML) is the standard language for creating web pages. HTML describes the structure and content of a web page through its markup. Web Browsers use an HTML document to render the content of a web page [35]. Since this language is the standard, it was a natural choice for our project to have the structure and content of the web interface represented in HTML. An HTML document is returned by the server on an HTTP GET request to the root of the SBACS server URL, and is then rendered by the client's web browser. Figure 19 shows the HTML of the SBACS web site home page, along with the page rendered by a web browser.

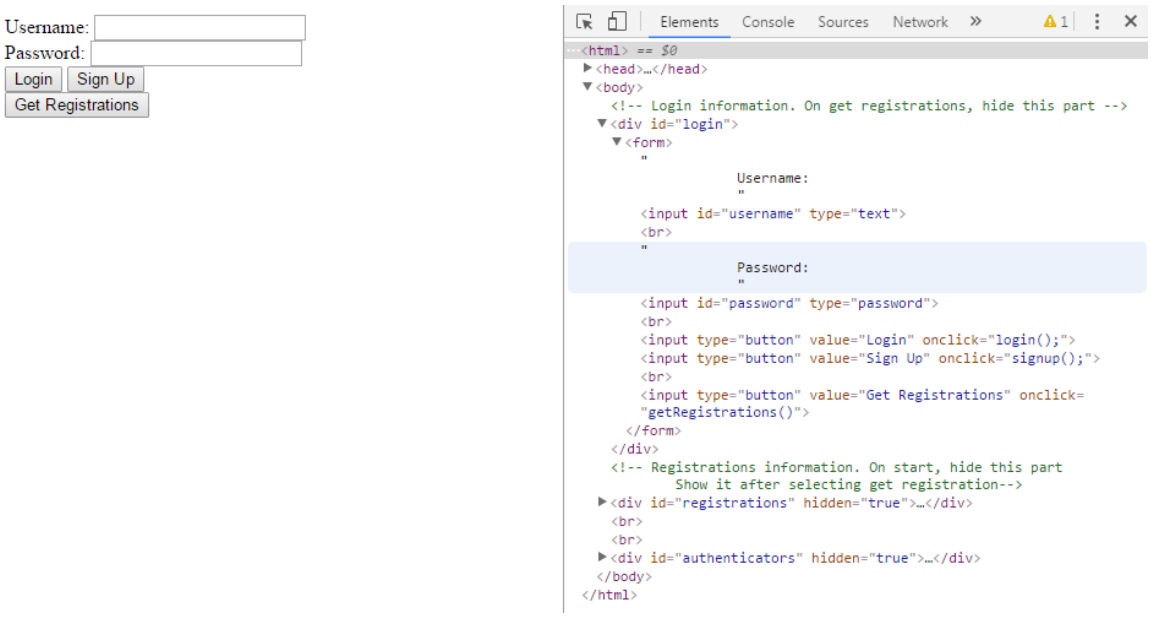

Figure 19: Rendered SBACS home web page (left) and HTML of web page (right)

5.5.6 JQuery

The structure and content of the web interface are statically defined using HTML (as discussed in section 5.5.5), but this will not be an interactive web page without the use of JavaScript. JavaScript allows for functionality to be added to certain actions performed by the user while interacting with the web page. In this project, we use JavaScript to dynamically modify the content of the web page and retrieve additional information from the server without requiring a page reload on the client side.

JavaScript has varying levels of support across different web browsers. This introduces a problem as different clients will have a different user experience, and some functionality may not work for all clients. JQuery is a library that helps to solve this problem by providing standard functions that are supported by many different web browsers [36]. There is also the added benefit of having functions that provide more functionality than the default JavaScript functions, making many operations easier on the programmer. For example, manipulation of the content and structure of the web page is made much simpler through standard JQuery functions [37]. The support across many browsers and the added functionality make this a very desirable

library to fulfill all of the needs of our web interface.

5.5.7 Single Page Application

The web interface is designed to be a Single Page Application (SPA). A Single Page Application is a web application that runs on a single web page, having no HTML page reloads [38]. This means that there is JavaScript performing all requests to the server to retrieve new information, and the JavaScript is also performing the manipulation of the web page to update the content and structure. This style of application was chosen in order to keep the server side processing of HTML to a minimum. Once the page has been returned to the client one time, there will be no more HTML processing on the server, as all subsequent information is requested through JavaScript. This reduces complexity on the server processing, as a static HTML file is always returned, and no actual processing of HTML is needed.

5.5.8 MySQL

To store data that is needed by our system, we decided to use a relational database. All members of the group have taken at least one course that has talked about relational databases, so we all had at least some basic understanding of this type of database. MySQL was chosen because it was known to members of the group, making it easy for everyone to make queries to the database whenever it was needed.

5.5.9 Relational Database Normalization

Database Normalization is used to efficiently organize tables in a database to eliminate redundant data and to maintain all information that exists in the data [39]. The end goal of normalization is to reduce the amount of data that is stored, without removing necessary information. There are various normal forms to measure the amount of normalization that has been used. The rules to meet first, second, and third normal form will be shown in this section, as our database tables meet these forms. Figure 20 shows the tables of our database and their relationships.

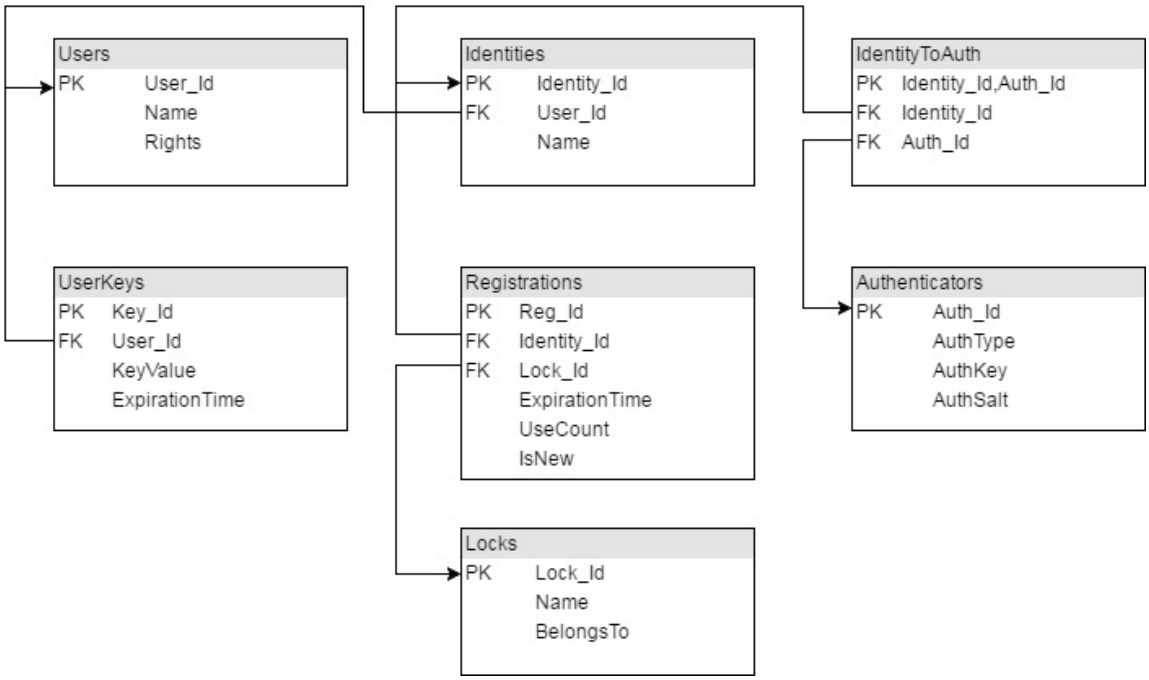

Figure 20: Database tables

To meet third normal form, first and second normal form must also be met. First normal form has three rules: data items must be defined and grouped in tables, there must be no repeating groups of data, and there must be a primary key for each table [40]. In our database we have many tables that are used to group data; one example is the Users table that groups data items such as User_Id, Name, and Rights. There are no repeating groups of data in any of the tables and each table has a primary key.

Second normal form requires that no partial dependencies exist [41]. This means that there cannot be data items that are dependent on a field that is not the primary key in a table. In our tables, we have ensured that no partial dependencies exist, so we meet second normal form. Third normal form requires that no transitive dependencies exist in a table, meaning that a data item does not depend on the primary key of the table [42]. Again, in our tables there are no transitive dependencies so we meet third normal form.

5.5.10 Authentication for Server API

Many requests to the server allow users to read, modify, and delete data, so we needed a way to ensure that users were identified and authenticated to allow certain requests to be performed. We used Keyed-Hash Message Authentication Codes (HMAC) to accomplish this goal. The user is able to login using their username and password, and upon successful login, they are provided with a key and user id. The key is then used to hash the body of subsequent messages. The client must also send the user id to identify who is trying to make the request. When the server receives the request, it will hash the body of the message as well using the key that it has stored for that user. If the hashed body that is provided is the same as what the server gets from hashing the body, then this means that the user is authenticated. The provided user id is used to determine which key the server will use to hash the message, and is also used to determine whether or not the user is authorized to make the request they are attempting to make.

5.5.11 Storing Authenticator Data Securely

In order to protect user passwords from attacks, the database must not store the passwords in plain text. If the passwords are in plain text, an attacker could simply read the data and know the users password. To prevent this problem, we use salting, which allows us to store a hashed version of the data, along with a salt that is used to generate the hashed value [43]. To compare a given password by the user to the value that is stored, the server must retrieve the salt and use it to hash the given password. The hashed value in the database should then match the hashed given password, meaning that the user has provided the same password. With this technique, the data is never stored in plain text so it is less vulnerable to many types of attacks.

5.5.12 Sending Data Securely Over the Network

The Hypertext Transfer Protocol (HTTP) is the standard protocol used to send requests on the internet [44]. In many situations, these requests need to contain sensitive information, such as passwords or personal information about the user. Over

this HTTP connection, this data cannot be sent securely, as there are security vulnerabilities in HTTP. In order to send this information securely, HTTPS is the common solution. In order to ensure the information is sent securely, there is a layer of authentication and encryption between the client and server for every HTTPS request that is made [45]. This ensures that only the sender and intended recipient are able to view and modify the contents of the message. In our project we have ensured the use of HTTPS wherever it is necessary to protect our sensitive information.

5.5.13 Design Patterns

In order to handle incoming requests to the server, we use a Front Controller pattern. In this pattern, all request to the server are handled by a single controller, the Front Controller, and the requests are sent to the appropriate controller to be handled [46]. As can be seen in Figure 21, the requests will all go to the FrontController. This controller will determine, based on the path and method of the request, which controller should be called to complete the request. The request will then be processed and a correct response will be generated by the controller that is called. For example, a request to login would be sent to the LoginController, whereas a request to access a lock would go to the AccessController.

The Facade pattern [30] is used to initially sign up a user for SBACS. There is a single request to the server to sign up, with a given username and password. This request will be processed by the Front Controller and then sent to the SignupController. Figure 22 shows the class diagram of how the SignupController interacts with multiple other controllers. The sequence of operations that are called can be seen in the sequence diagram in Figure 23. The SignupController will call the UserController to create a new User with the given username, then will call the IdentityController to create an Identity for the new User, and finally will call the AuthenticatorController to create an Authenticator that is a password with the given password for the User.

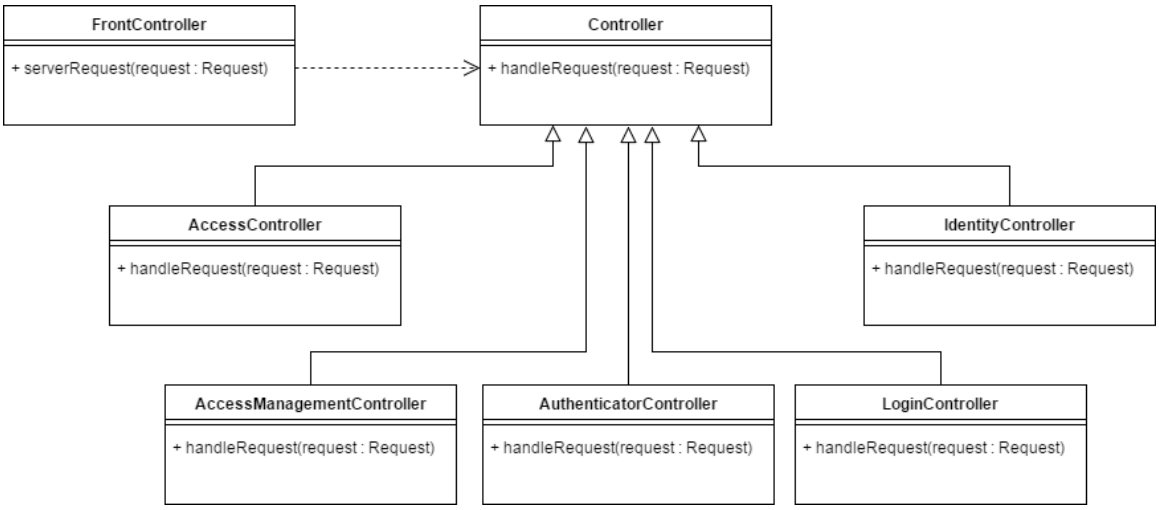

Figure 21: Front controller class diagram

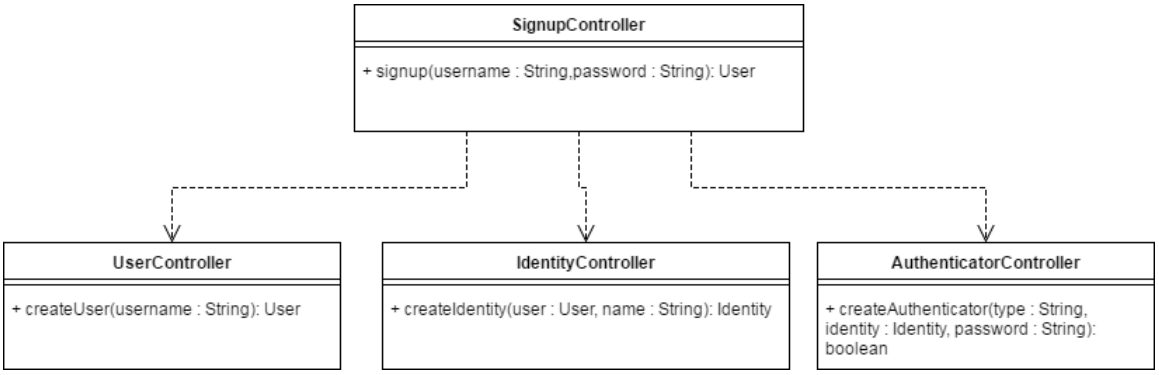

Figure 22: Facade class diagram

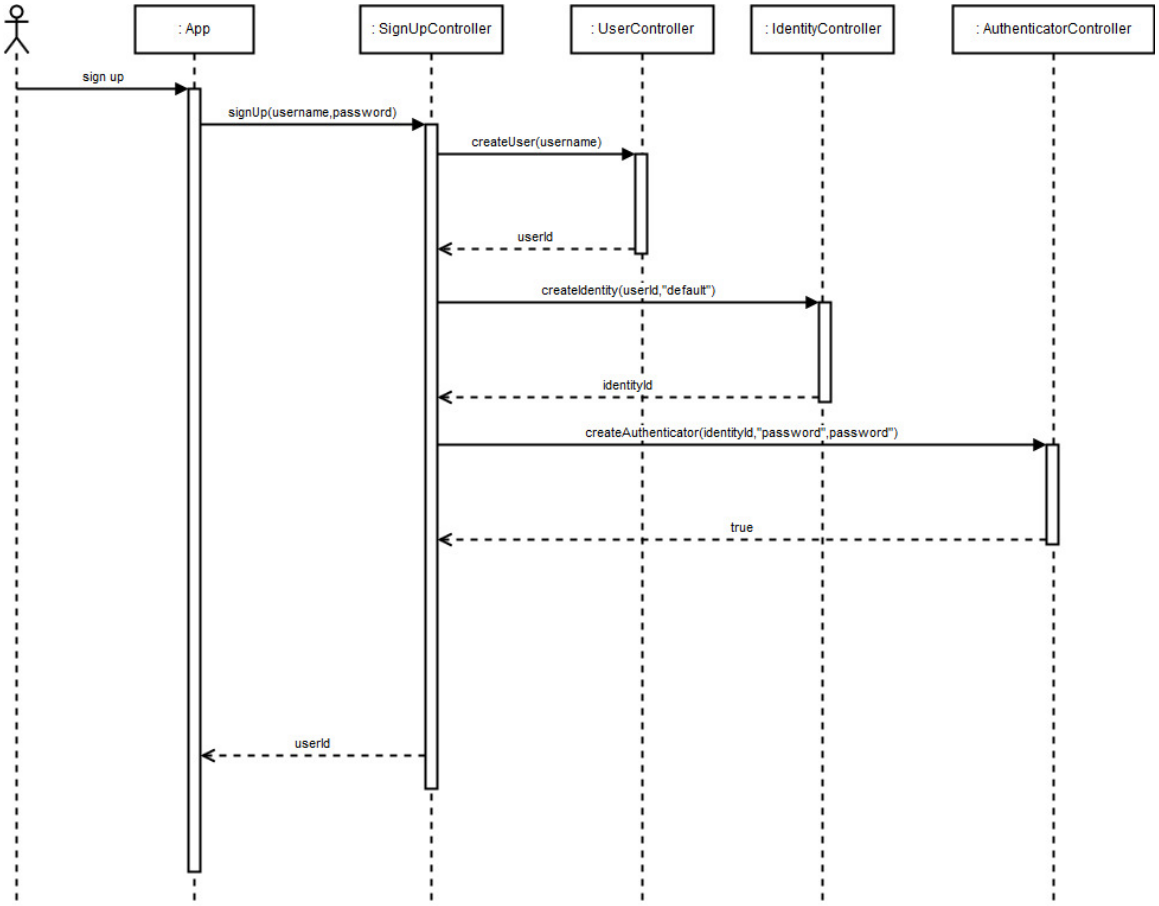

Figure 23: Facade sequence diagram

5.6 Poster Fair Demonstration

An end unit demonstration was prepared for the Systems and Computer Engineering poster fair. Since all of the SBACS hardware was prepared for then, all that remained was to find a storage container to lock using the SBACS. The container selected for the demonstration was a wooden bedside table with a drawer.

Firstly, to reveal the hardware component of the SBACS, the wood behind the drawer was removed, and replaced with a plexiglass panel. The drawer was shortened by two inches, in order to accommodate the SBACS hardware, which was affixed to the rear of the unit. The Raspberry Pi was attached to the plexiglass in the corner of the unit, and a hole was cut in that side of the unit to make the USB and Ethernet ports accessible. An cube-shaped enclosure was designed to have through-holes for the LEDs, as well as to protect the PN532 NFC shield. The PN532 was screwed to the top of the end table, and the LED enclosure covered most of the PN532, so that only the antenna was revealed. This hid the wires connecting the PN532 to the Feather, giving the unit a cleaner look, and protecting the NFC module from tampering.

Figure 24 is a photograph taken of the inside of the end unit, with the drawer removed to reveal the hardware layout. The outside of the unit is shown in Figures 25 and 26, with Figure 26 focusing on the status indication LEDs.

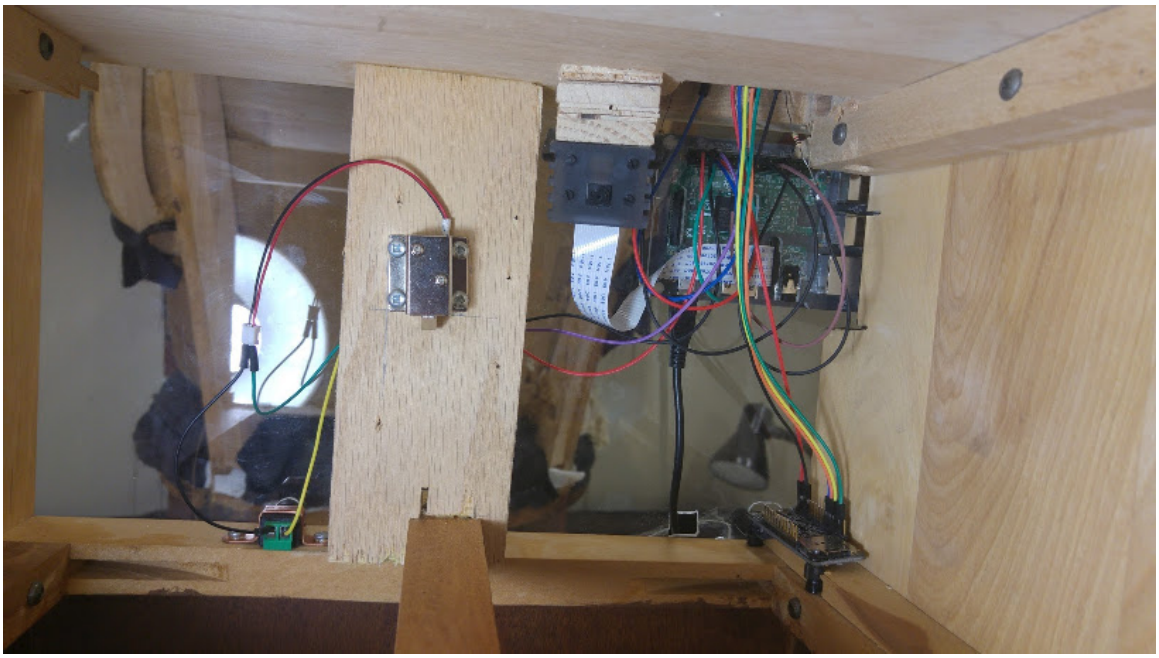

Figure 24: Inside view of the end unit demonstration

Figure 25: Outside view of the end unit demonstration

Figure 26: Outside view of the end unit demonstration, focused on the LEDs and NFC module

Chapter 6

System Implementation

This chapter describes the steps taken to implement the SBACS. The tools used along with the design patterns implemented are described in detail, along with any design decisions that had to be made or changed when the need became apparent. This chapter also outlines all of the protocols used to facilitate communication between the various devices that make up SBACS.

6.1 Android Application Development

The Android application was developed using Android Studio [47] exclusively, and is primarily composed of Java classes, XML documents, and Gradle build files. Android studio was selected since it seemed to be the natural tool for writing an Android application, and didn't have any negatives with respect to our requirements. Gradle [48] was chosen as the build tool as it is naturally integrated with Android studio.

6.1.1 Server Interfacing

Connections to the server were managed through the android Volley API. Volley provides a simple interface over which HTTP and HTTPS messages can be sent [49] [50]. Standard operation of Volley follows a simple procedure: first a RequestQueue is set up, and then various Requests are enqueued. These requests are then dequeued by

the RequestQueue's thread, which then creates an HTTP/HTTPS connection. The response that comes back over the connection is handled by another thread.

The code for handling these responses is put in a class that extends a Response.Listener of a given type. The Response.Listener class provides methods to handle HTTP responses. Since these classes have just a few relatively simple methods, anonymous classes were used in all cases. Further, we decided that handling the parsing should be done as safely as possible, so all responses were first parsed simply as Strings. Then, the response was attempted to be parsed as a JSON object of the type expected from the server. If that parsing failed, the message was instead considered an error, and handled from there. A high level example is shown in figure 27 in the particular case of logging in to the application. Note that in this figure, the getBody call that the StringRequest performs is actually asynchronous. This shows how much the operations in Volley are performed asynchronously on the fly, and if these were to take too long, the message would be sent by the RequestQueue before the StringRequest finished. In this case, this was not particularly important, since the only operation performed was converting a JSON object containing a few bytes to a byte array, but had to be considered in other cases.

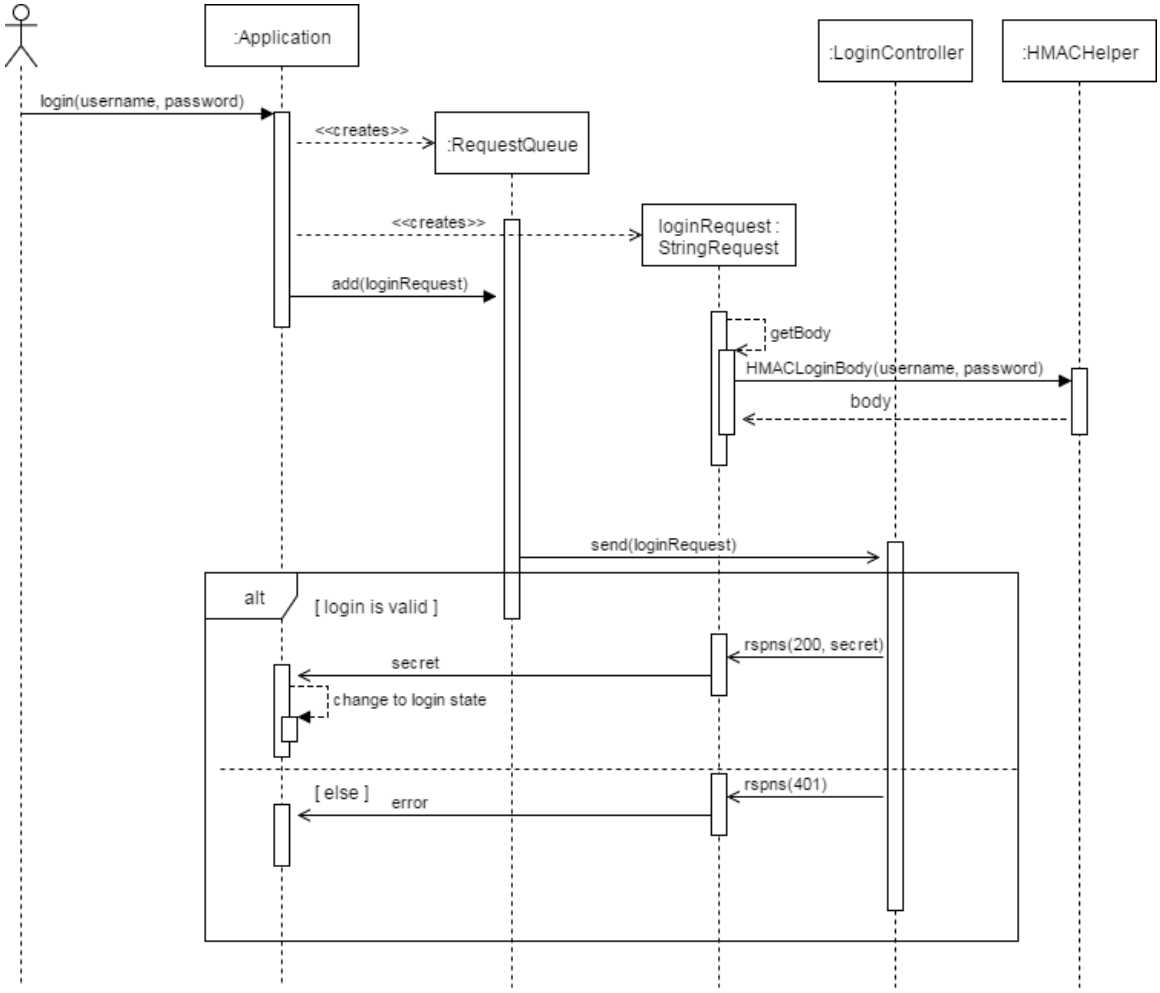

Figure 27: Login sequence diagram demonstrating Volley

6.1.2 HMAC

Since the server used HMAC to handle security issues, the application needed to make frequent use of the headers that the server expected to handle authentication. For this reason, a helper class called HMACHelper, was created. This class provided methods that were commonly used by all requests about data particular to the current user. Most notably, HMACHelper provided a method which calculated the secret using the same algorithm that the server would use. To reiterate how HMAC functions, the body of the message is hashed using a private key shared by the server with the application. When the application sends data that requires authentication, the server checks that the secret value provided by the application matches what it calculates using the body of the message as well as the private key associated with the user. These private keys are generated by the server when the user logs in to the application, and expire over time, creating the notion of a session.

The precise algorithm used for the hashing function matches the decision made for the server. This is necessary, since otherwise the secret values would not match between the application and the server. The application relies on a version of the Password-Based Key Derivation Function algorithm using SHA256 [51] to be available on the Android device. The only workarounds for this added requirement would have been to implement the algorithm (specified in RFC 8018 [52] with test vectors in RFC 6070 [53]) ourselves, or by making use of a third party implementation. However writing it ourselves is heavily discouraged as any minor error could result in a security breach, and we did not want to have to worry about any intellectual property issues of using a third party library. In addition, the number of iterations and the length of the resulting key had to be synchronised between the server and the application to ensure the values match.

A simplified demonstration of this process is shown in figure 28. This figure shows what the application does when a user requests the information related to all the locks that they have a registration for. Several steps, such as the Volley RequestQueue, have been abstracted out for the purpose of simplifying the message of the diagram. For more details about volley, see section 6.1.1. For more information on how the server handles HMAC, see section 5.5.10.

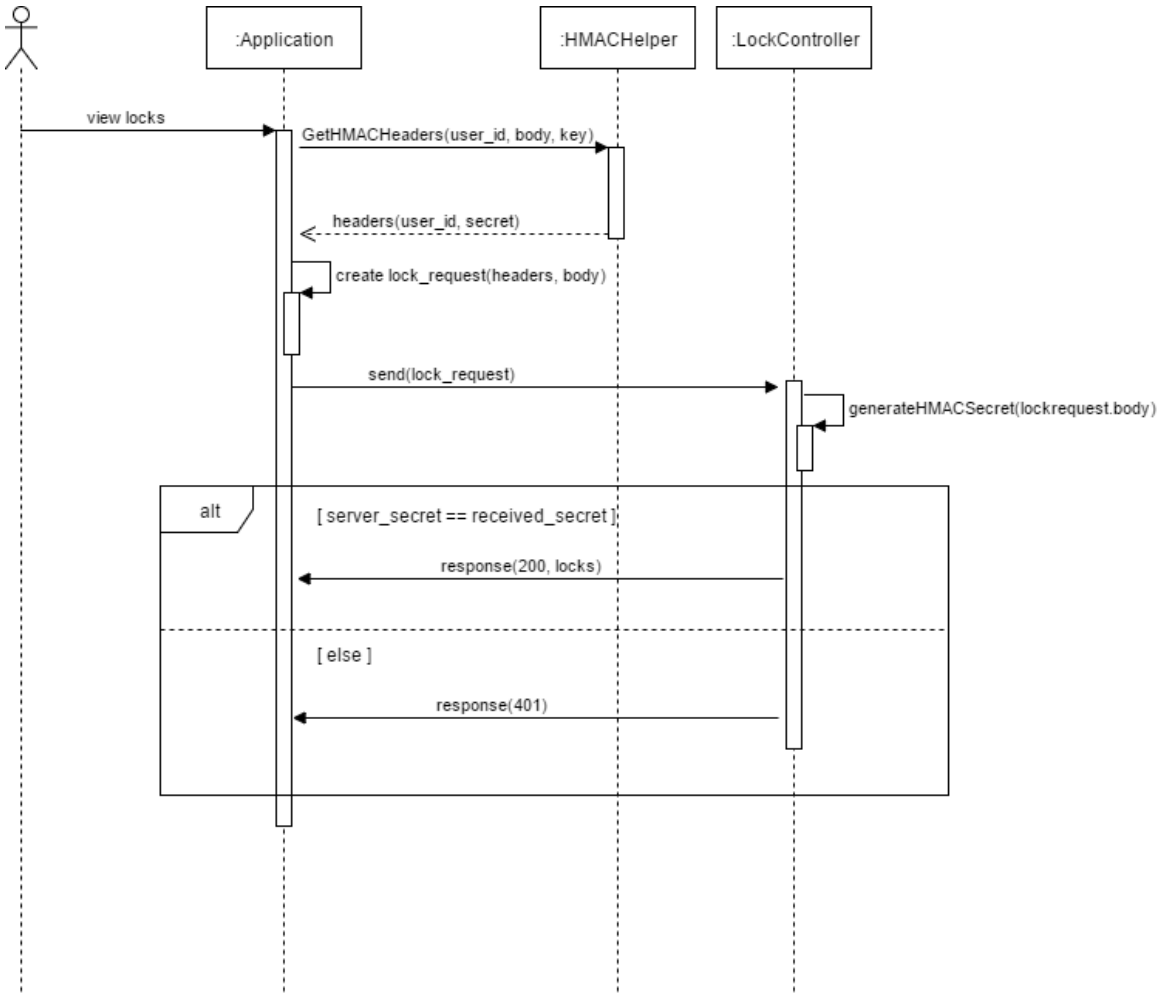

Figure 28: Sequence diagram of the application requesting sample data after logging in

Sometime after this was implemented, an update to the Android operating system resulted in the algorithm used for key derivation to no longer be supported. Since this occurred at a time near to a project deliverable, we felt that the fastest way to correct the problem was to change the algorithm used to rely on SHA1 instead, and then later to use SHA256 again but with a different algorithm. This is further explained and explored in section 7.2.2.

6.1.3 Entity Representation

The application needed to be able to display data to the user that came from the server and database. These data are represented in the MVC design of the application. Since the database is relational, the model classes that represent the constructs stored in the database, such as identities, authenticators, locks, and registrations, were all passive entity classes whose only real role was to contain data.

Each kind of data that was of interest to the user was given its own Activity, which is similar to the notion of a page in a web application. These Activities were all primarily composed of a ListView object, which displays a list of data. This list was configured with an Adapter class which extends the general ArrayAdapter¡T¿ class. These adapters were used to convert the model classes into a view. The adapter classes were composed of an XML file which described the layout of the view, as well as a Java class which provided the methods by which the data could be used to fill in the layout.

6.2 Hardware

The hardware portion of the project consisted of three components: the lock control circuit, the lock control software, and the authentication modules.

6.2.1 Lock Control Circuit Implementation

Selecting the device to run the core process was the first step in implementing the lock control circuit. A single-board microcontroller was the first option considered,

as it had the potential to be lower in cost than a single-board computer, given that it would fulfil all of the requirements for the core circuit.

Since driving the LEDs and the electrical lock only requires digital output, no analog pins would be necessary. This would lower the cost of the hardware, as digital-only boards tend to be cheaper than analog-capable boards. While the single-board microcontrollers considered had optional add-ons which could add Wi-Fi or Ethernet capabilities, they would not have been a sufficient solution to run the core process, as they were lacking a sufficient number of USB ports, and the overhead for implementing the core would have been too high. Because the microcontrollers do not run an operating system, essential primitive operations such as file opening, checking device attributes, and reading/writing to devices would not be available. For these reasons, a single-board microcontroller was not selected to run the core process for the lock control circuit.

A Raspberry Pi was ultimately selected as the computer to run the core process. The Pi 3 Model B has four USB ports, built-in Wi-Fi capability, runs Linux, and has forty pins for general-purpose IO (GPIO). These hardware features made the Raspberry Pi the best out-of-the-box solution to run the core process. The GPIO pins provided the digital outputs for driving the status LEDs and the lock. Interfacing with serial devices is simple to implement using Linux, as the devices are represented as files, and can be read from and written to like files.

The lock used in the proof of concept was a solenoid lock. This lock required 9V to 12V DC, and drew 650 mA when operating at 12V.

6.2.2 Lock Control Circuit

The next design problem to consider was the means to control the status of the electric lock. The simplest locks available, solenoid locks and electric strike locks, do not include any control mechanisms. They are either powered on, to disengage the lock, or powered off and locked. Therefore, it was necessary to implement a circuit to control the power to the lock, with the switching being driven by one of the Raspberry Pi's GPIO pins. There were two options considered for this: a circuit

using a transistor and diode to switch on the lock, or a relay.

The transistor and diode circuit in Figure 29 shows the first possible implementation of a lock control circuit. One of the Raspberry Pi's GPIO pins is connected to the base terminal of the TIP120 NPN transistor, which allows the transistor to act as a electronic switch controlled by the Pi. The ground input of the solenoid lock is connected to the collector terminal, while the +12V input is connected to a diode which connects to the collector terminal. Finally, the emitter terminal is connected to a ground rail. The diode is in place to prevent any damage to the system when the solenoid is turned on, as solenoids will produce a voltage spike known as "flyback", which occurs when the current to the solenoid experiences a sudden change [54].

When the base is set to off by the Raspberry Pi, the base-emitter voltage V_{BE} will be 0 as the base and emitter terminals are at 0 V, putting the transistor in the cut-off state. In this state, the transistor does not allow current to flow from collector to emitter. When the Raspberry Pi turns the base on, $V_{BE} = 3V$, as the base voltage is now 3V. Since the TIP120 requires $V_{BE} = 2.5V$ for saturation mode, this means that effectively, the switch is turned on and current may flow from collector to emitter.

One of the issues with the transistor and diode circuit was that there were concerns that a heat dissipation mechanism would be required to safely operate the transistor. The TIP120 transistor can safely dissipate a maximum power of 2 Watts [55], and can reach a maximum operating temperature of 150 degrees Celsius. Preliminary calculations showed that power dissipated by the TIP120 would likely reach or exceed 2W. Adding a fan to the system could have removed this risk, but at the expense of compromising the overall security of the hardware. The lock control circuit, in a real-world scenario, should be kept enclosed and hidden away to prevent break-ins. Adding a fan would compromise both of those points.

For these reasons, the relay was chosen for the lock circuit implementation. The relay would not require a heat dissipation mechanism, in addition to physically isolating the power circuit for the lock from the lock control circuit. Figure 30 is a circuit diagram of the internal layout of a relay.

Internally, a relay is very similar to the transistor and diode circuit, with a mechanical switch added. The relay has three inputs, and three outputs. The inputs are

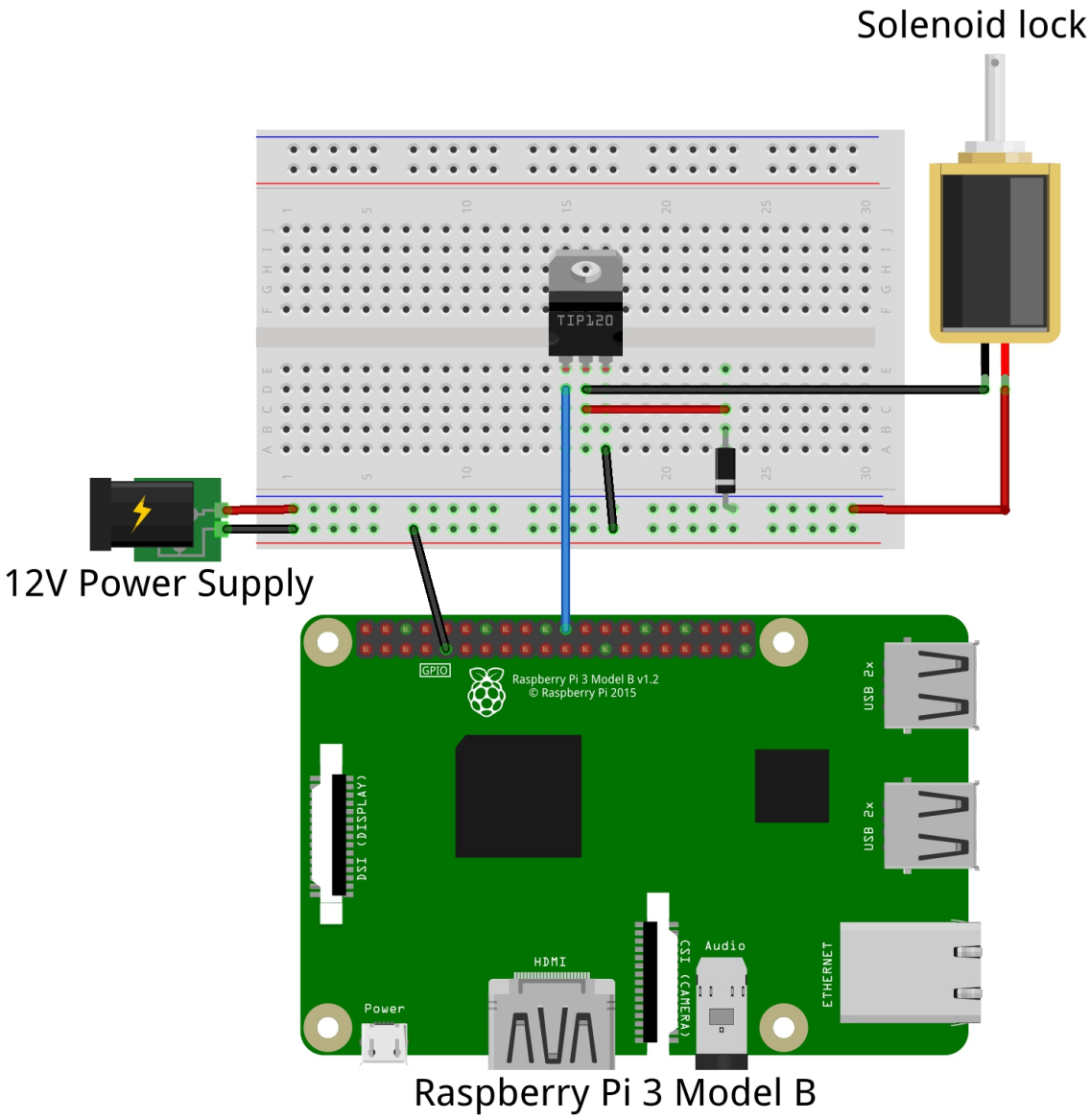

Figure 29: Transistor-diode lock control circuit

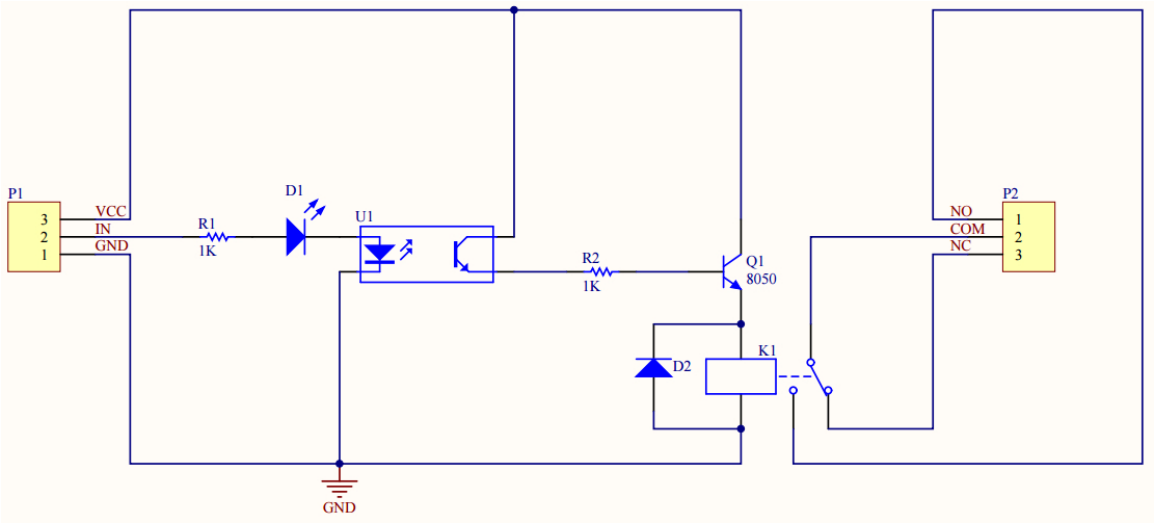

Figure 30: Relay circuit diagram [56]

a positive voltage (V_{CC}), ground (GND), and an input signal (IN). The outputs are a normally-closed terminal (NC), a common terminal (COM), and a normally-open terminal (NO). In the implementation of the lock control circuit, V_{CC} is provided by the 5V pin on the Raspberry Pi, and the IN signal is provided by a GPIO pin.

In Figure 30, diode D1 and the assembly U1 are status LEDs. Diode D1 indicates when the relay is powered on, while the assembly U1 indicates whether the relay's internal switch is flipped to the NC terminal (LED off), or to the NO terminal (LED on). The transistor in assembly U1, combined with the transistor Q1, control whether current flows to the solenoid K1. When IN is high, the base voltages at both transistors sets them to saturation mode, so current will flow through solenoid K1. Diode D2 protects the relay circuit from the kickback voltage of solenoid K1. The mechanical switch, or armature, is indicated by the dashed line connected to solenoid K1. The armature's position is controlled by K1. When the relay is powered off, the armature is positioned so that the NC terminal and COM terminal are connected. When the relay is powered on, the armature is attracted by K1 to switch to connecting the NO and COM terminals.

The final layout for the lock control circuit is demonstrated in Figure 31. The 12V terminal of the solenoid lock is connected to the NO terminal of the relay, while the

12V terminal of the DC power supply is connected to the COMM terminal. When the IN signal from the Raspberry Pi is low, the 12V terminal of the lock is disconnected from the power supply, so the lock is powered off and locked. When the IN signal is high, the 12V connection is made, powering on and unlocking the lock.

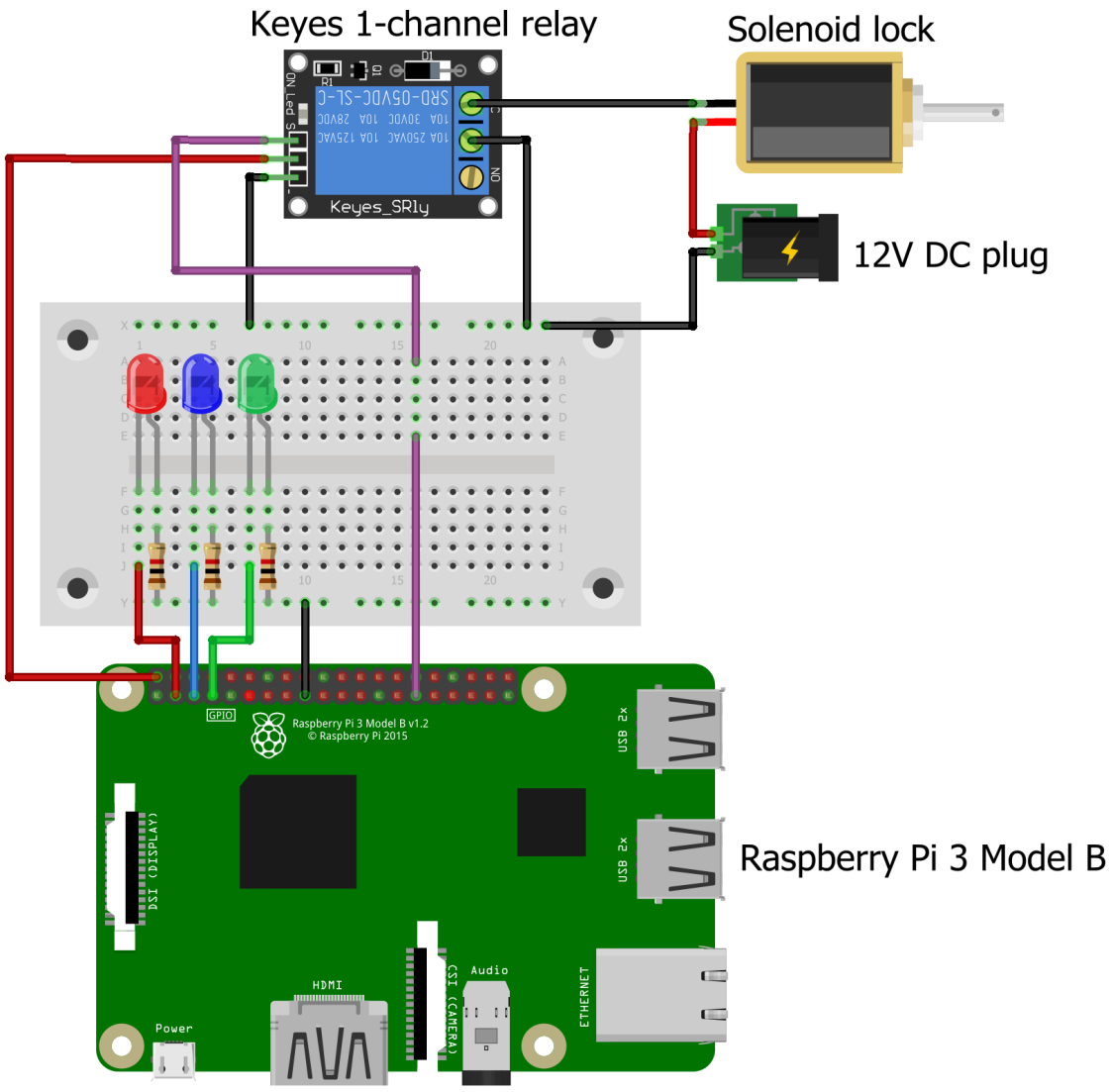

Figure 31: Lock control circuit

6.2.3 Lock Control Software

The lock control software, running on the Raspberry Pi, was written in C++. C++ was selected primarily because the software for the authentication modules could only be written in C++, and using the same language as the modules would allow the core process to share classes with the modules. The second major reason for selecting C++ was that it was an object-oriented language, so the design patterns mentioned in section 5.4.2 could be implemented as described.

When the system is first run, it performs a series of self-checking steps. First, it verifies that it has internet access and can reach the SBACS cloud server. If the connectivity check is successful, it then checks each of the USB port devices to verify if the device is a SBACS authentication module. If at least one authentication module is connected, then the system enters a polling loop. It will poll the authentication modules to check for data, and when there is data available, collect it into an access request and send that request to the server. If the server response code is OK, then the lock is disengaged for three seconds. This setup sequence is demonstrated with the specific function implementations in Figure 32.

The final implementation of the lock control hardware did not deviate significantly from the design. The major change between the initial design and the final design is the removal of the VideoStream class, which was not necessary for the end unit. The full class diagram of the lock control software, Figure 50, is available in Appendix A. The classes are divided into four categories describing their use:

Firstly, the Entity classes used for communication are the Packet and its subclasses, and the AuthenticationToken class. These classes are represented in UML in the class diagram, Figure 33. The Packet classes are used for communication between authentication modules and the main process. The code for these classes is shared with the authentication module's source code. EncodedPacket objects are used when one device has a byte buffer to send to another over USB. DecodedPacket objects are used to translate a stream of bytes received over USB into a byte buffer containing only relevant data. Packets are described in more detail in the Section 6.2.5, which describes the protocol communicating over USB. AuthenticationToken objects are created by the main process to internally represent a complete authentication

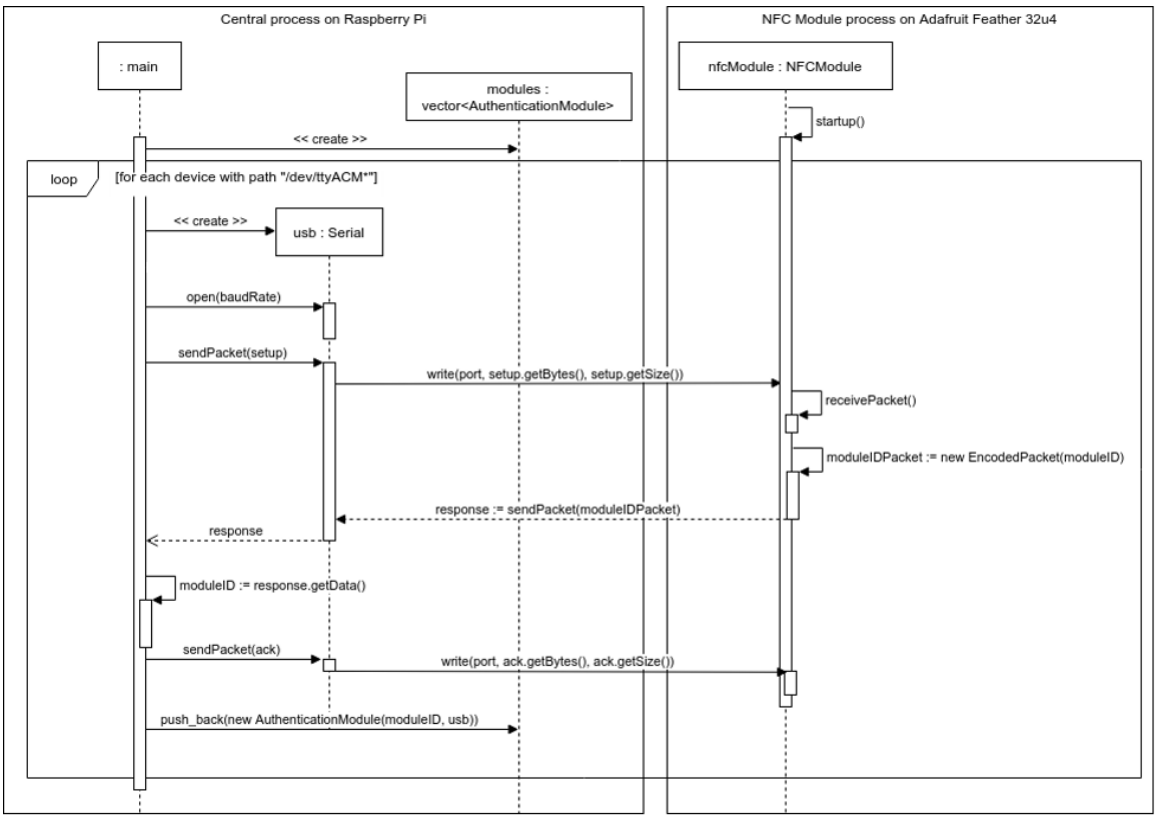

Figure 32: Hardware start up sequence diagram

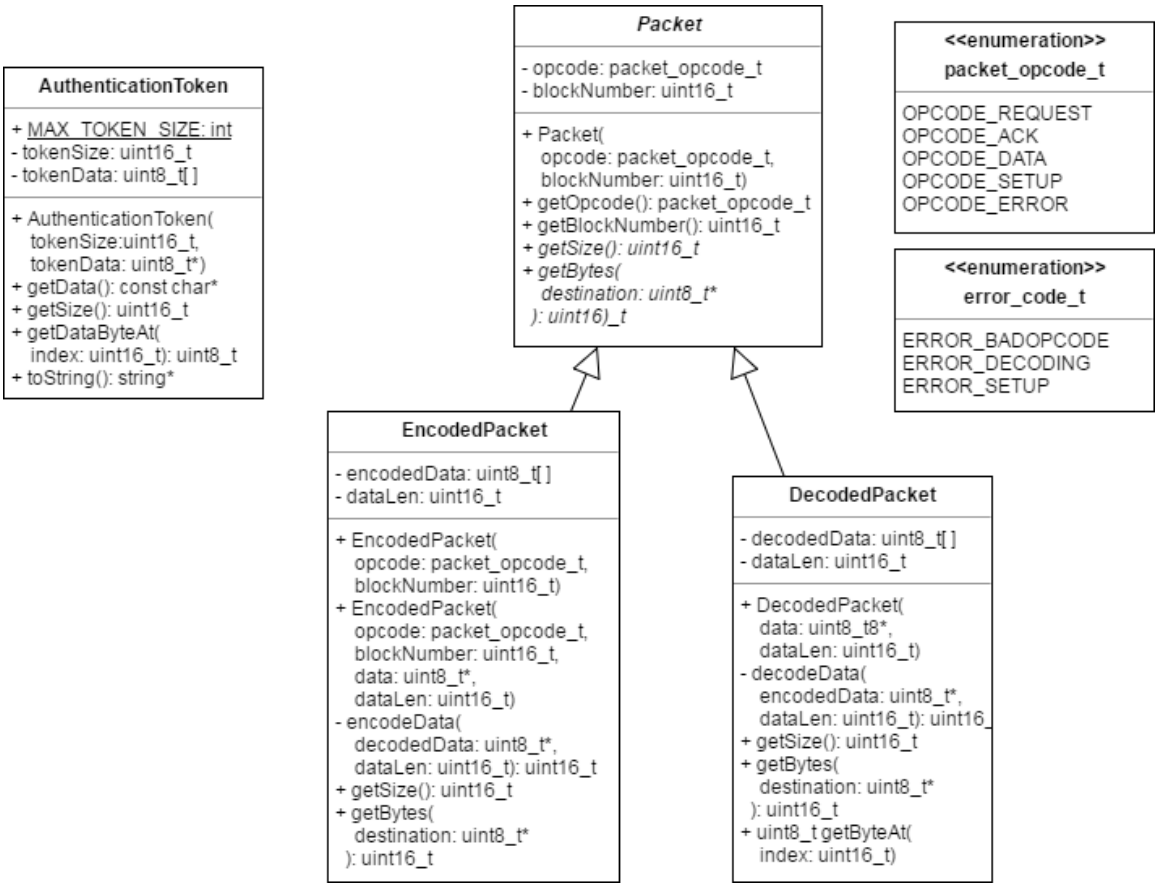

Figure 33: Hardware entity classes

token that has been received from a authentication module. Because some modules have raw bytes as data and others may use ASCII strings, methods are available to represent AuthenticationTokens as either structure.

The second grouping of classes performs communication using the aforementioned Entity classes, pictured in Figure 34. The AuthenticationModule class is an abstraction of the authentication modules connected to the Raspberry Pi. This class associates the module's string identifier with a reference to the serial line of the module, and keeps a reference to an AuthenticationToken if one was received. The ServerConnection class is used for communicating with the web server. To accomplish this, this class uses the libcurl C library. With libcurl, all of the work done to execute a HTTP request is handled by the library, so the ServerConnection is only responsible

for preparing the request, including setting the headers, and generating the request body.

The third grouping of classes provides abstractions of various hardware components. The GPIOPin class is used for controlling the GPIO pins of the Raspberry Pi, which for this project only needed to be set as output and be written high or low. The Serial class represents a USB port on the Raspberry Pi, and provides the mechanisms needed to interact with the port. The most notable methods in this class are receivePacket() and sendPacket(), which use the Packet classes. Refer to Figure 35 for more details.

Finally, to define all of the behaviour of the lock control software is the main class, which deserves a category of its own. The main class is the mediator between the various AuthenticationModule objects, the ServerConnection, and the GPIOPins. Its various functions are pictured in the class diagram, Figure 36. Grouped with the main class is the Debug class, which is used for printing debug messages to the console when testing the system. See section 7.2.1 for more detailed information on the Debug class.

Due to the number of .cpp files that resulted from this design, a makefile was written to automate the compiling and linking of all of the classes. The makefile has four options for building the lock control software. Firstly, a "make" with no arguments will compile and link all of the previously mentioned classes, outputting a single executable file. A "make debug" performs the same actions, but defines the DEBUG global variable to enable debug messages to be printed to the terminal when the lock control software is executed. Using "make test" compiles the Packet classes and a unit test for them, and then runs the test. Finally, "make clean" simply removes any executable files and intermediate object files that may have been left over from an unsuccessful compile.

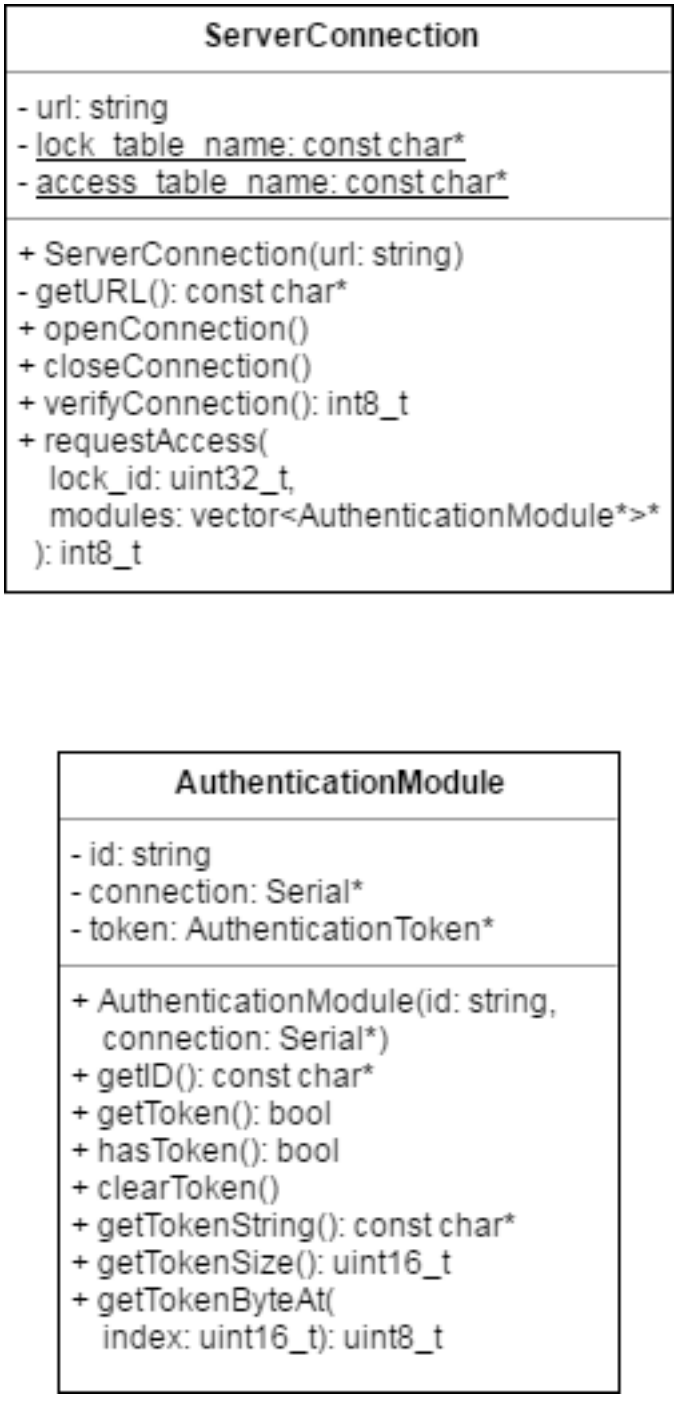

Figure 34: Hardware communication classes

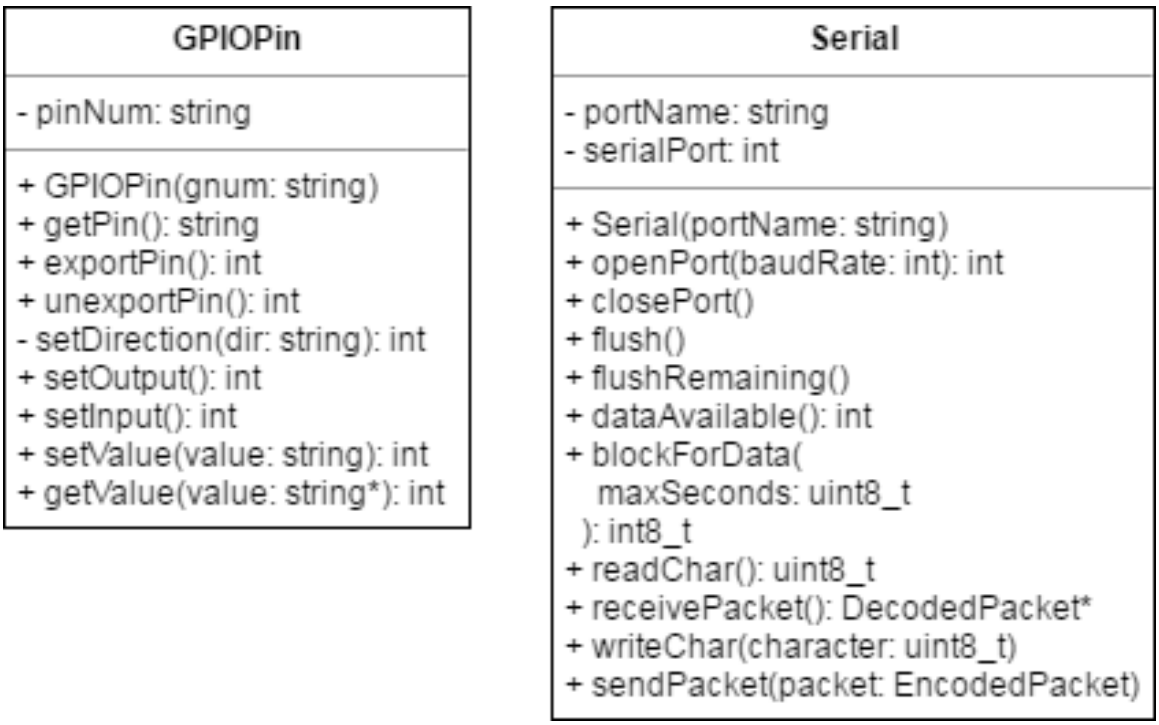

Figure 35: Hardware abstraction classes

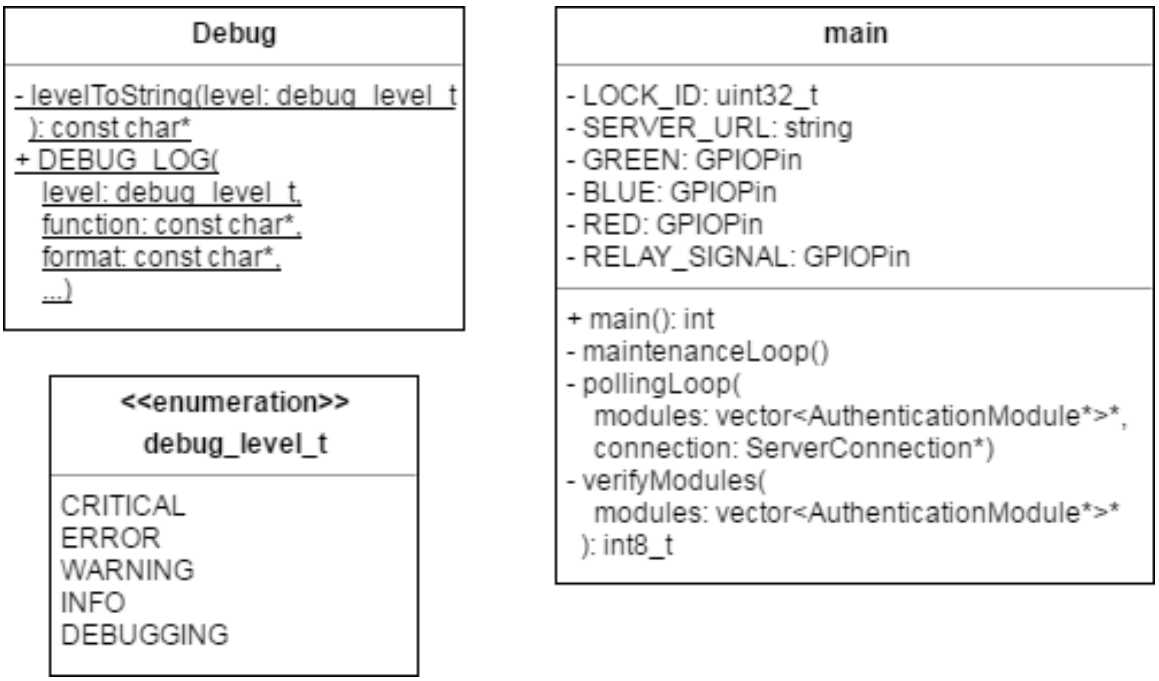

Figure 36: Hardware top-level classes

6.2.4 Authentication Module Implementations

The authentication modules in this project were powered by Arduino IDE-compatible microcontrollers. Because the officially supported Arduino library is written in C++, the software for the authentication modules had to be written in C++.

Applying the design patterns outlined in Section 5.4.3 was simple, and the actual implementation did not significantly deviate from the design. The AuthenticationModule superclass grew to accommodate more requirements, such as adding a string identifier to the module so that the type of token can be identified. It can be seen from the class diagram, Figure 37, that the sendData() template method was used, with the pure virtual method getData() rendering the class abstract. Additional methods for communicating with the Raspberry Pi, sendPacket() and receivePacket(), are used by startup() and sendData() to convert bytes received or sent over the serial line into appropriate variations of the Packet object.

One notable change from the Entity-Controller-Boundary design from Figure 17 was that separate objects for the Module-Specific Data Handler and the Module Controller were not necessary, as the modules' software was simple enough that the Boundary and Controller functionality was implemented in the AuthenticationModule class. In addition, the Entity class generated by the AuthenticationModule class was the EncodedPacket class, instead of a "Authenticator" as it was named in Figure 17.

Figure 38 shows the sequences of events from a user interacting with an authentication module, to transmission to the core process, to sending an access request to the server. This sequence uses the NFC module for example, but can apply to any other module.

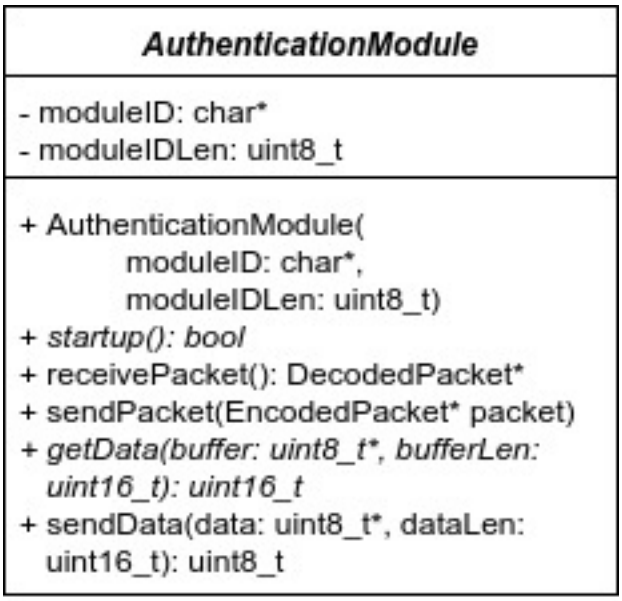

Figure 37: Authentication module class diagram

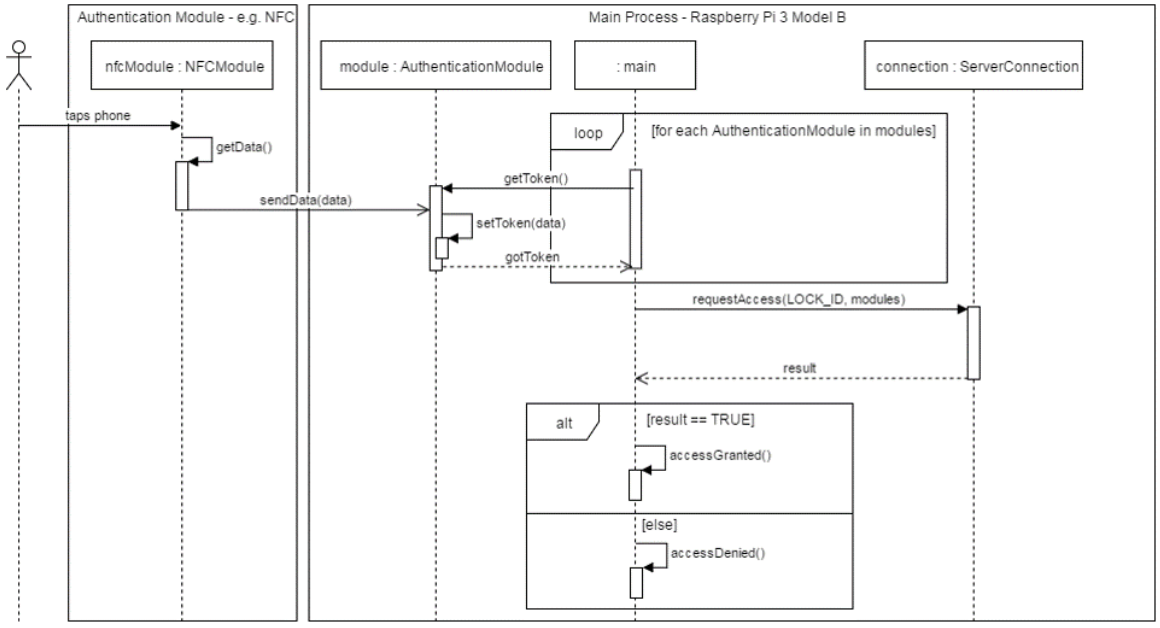

Figure 38: Hardware access attempt sequence diagram

NFC Module

The module that was the primary focus of this project's proof-of-concept was the NFC module. The module uses an Adafruit PN532 NFC shield to read NFC data from an Android phone. This model of NFC shield was selected because it was the only one available to be purchased in Ottawa that supported NFC communication with Android phones. The PN532 is driven by a Adafruit Feather 32u4 single-board microcontroller. The Feather was selected because of it uses 3V logic, matching the PN532's 3V logic. The wiring of the NFC module hardware is shown in Figure 39.

The PN532 shield has three modes of operation, specified by the SEL0 and SEL1 pins on the left side of the shield. These are Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transfer (UART), and Interrupt Request Line (IRQ). SPI was selected as the mode of operation, as it was simple to implement in software, and required only five pins to communicate with the Feather. Additionally, it is the only mode that supports software polling, as the UART and IRQ modes are for asynchronous communication and require interrupt handlers. The PN532 shield is run in Serial Peripheral Interface (SPI) mode by wiring the SEL0 pins "off" and the SEL1 pins "on".

To support the synchronous data communication, the serial clock (SCK) signal is sent from the Feather to the PN532 over the blue wire in Figure 39. This ensures that the Feather and the PN532 are operating at the same clock frequency, so bits sent by

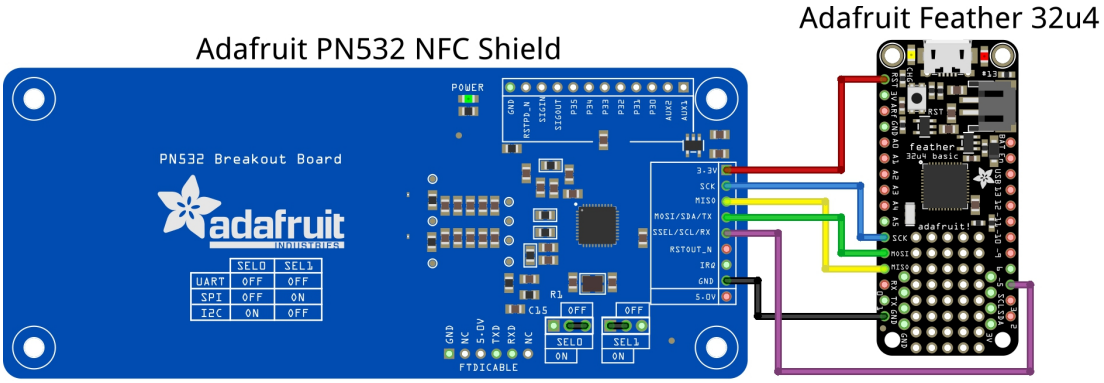

Figure 39: NFC module wiring diagram

one device are received by the other device at the same rate. Data is sent from the Feather to the PN532 using the master-in-slave-out (MISO) signal, the yellow wire. Conversely, the green wire is for data from the PN532 to the Feather, the master-out-slave-in (MOSI) signal. Finally, since SPI mode supports having multiple slave devices, the slave select (SSEL) signal is wired to pin 5 on the Feather. The Feather will raise the SSEL signal when it is interacting with the PN532, such as for polling.

For the software, the NFC module used the SeeedStudio PN532 library, as the Adafruit PN532 library did not support communication with Android devices. Figure 40 demonstrates the class diagram for the NFC module. Note that the module overrides the startup() method from AuthenticationModule, as the PN532 must also be set up, with an ERROR packet sent during startup() if there is an issue with it. For more information on how the NFC module interacts with Android devices, refer to section 5.2.3.

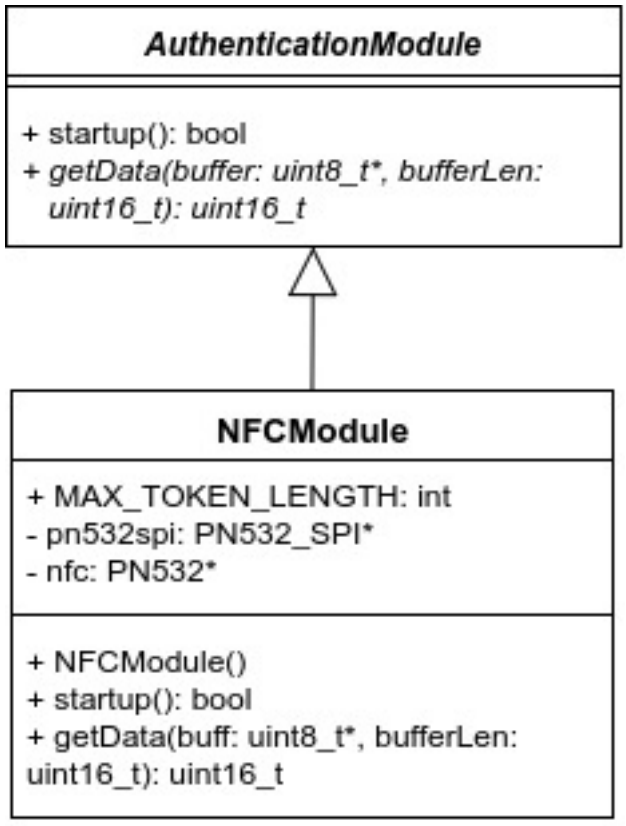

Figure 40: NFC module class diagram

PIN Module

The PIN module is a simpler module than the NFC module. It uses a 12-button matrix, which uses four output pins to specify the row of the button, and three output pins to specify the column. As pictured in Figure 41, the matrix output pins are wired directly to seven input pins on the Arduino.

The keypad matrix has seven output pins, four row pins and three column pins. When a button is pressed, it activates the corresponding row line and column line, which the Arduino can read to determine the coordinates of the button the keypad. The Keypad library included with the Arduino IDE maps the PIN pad to a 2-dimensional array, and handles mapping the row/column inputs to a character. In addition to this, the Keypad library handles key debouncing.

Since the Arduino has a built-in digital pin for LED debugging, a red LED was added as a status indication when entering a PIN. This LED would turn on when the PIN module is accepting input from the user, and would blink when sending the user's inputted PIN to the core process.

The PIN module accepts input from every button, except for the asterisk button. Users enter their PIN using the numerical buttons on keypad matrix, and then press the pound ('#') button to confirm their entry and send it to the Pi. A five-second timer starts after each button press, so that if a user abandons entering their PIN, then that partial PIN will be de-buffered after five seconds.

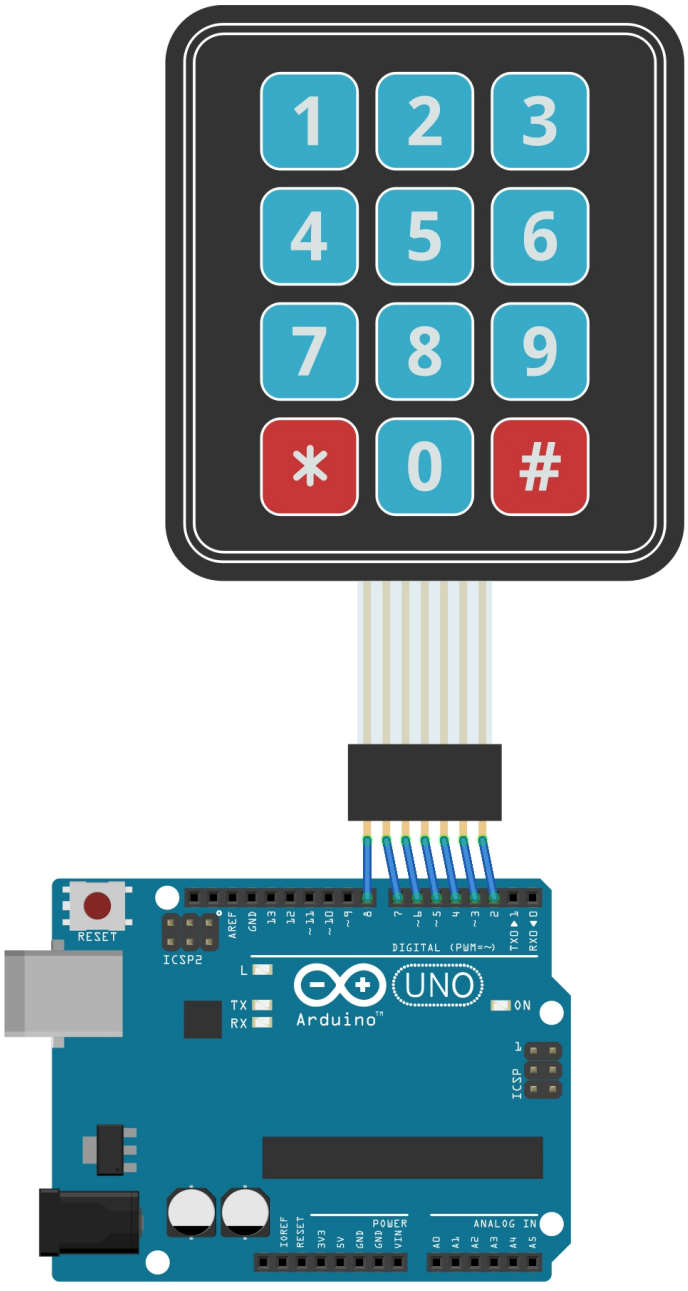

Figure 41: PIN module wiring diagram

6.2.5 Serial Communication Protocol

In order to distinguish actual communication data from noise on the serial line, a protocol was necessary to allow for error-free communication between the authentication modules and the Raspberry Pi. The protocol is a simple set of rules for formatting raw bytes into data packets, largely inspired by the Trivial File Transfer Protocol (TFTP), specified in RFC 1350 [57].

When a user interacts with an authentication module, the data they send is buffered in the module's RAM. Once an interaction between the user and the module is complete, the module initiates a data transfer with the Raspberry Pi. The authentication module divides its buffered data into 254-byte blocks, and sends them in sequence, as the Pi acknowledges them in sequence. The reasons for this design will be outlined in the remainder of this subsection.

Similarly to TFTP, the Serial Communication protocol supports five types of packets:

1. Request data transfer (REQUEST)
2. Acknowledgment (ACK)
3. Data (DATA)
4. Module set-up (SETUP)
5. Error (ERROR)

Packets start and end with a single byte known as the packet flag. The packet flag could have been any arbitrary byte, so 0x7E was selected. 0x7E is outside of the range of ASCII, so it would not commonly appear in the data. Following the starting packet flag is a Serial Communication Protocol header, which consists of two two-byte values: the packet opcode, and a block number. The opcode field defines which of the five listed types is the type of the packet, while the block number field's use varies depending on the packet type. All packets end with the packet flag.

REQUEST packets are sent by authentication modules to the Raspberry Pi, to notify the Pi that a full authentication token has been buffered by the module, and

0	1	2	3	4	5
FLAG	REQUEST		BLOCK		FLAG
0x7E	0x00	0x01			0x7E

Figure 42: Serial REQUEST packet format

0	1	2	3	4	5
FLAG	ACK		BLOCK		FLAG
0x7E	0x00	0x02			0x7E

Figure 43: Serial ACK packet format

is ready to send. The format of a REQUEST packet is shown in Figure 42. For REQUEST packets, the block number field denotes the number of DATA packets that the module will send. This allows the Pi to allocate a data buffer to match the size of the authentication token, instead of an arbitrarily large buffer, which is wasteful for smaller modules such as the PIN module.

The Raspberry Pi acknowledges REQUEST and DATA packets using ACK packets. Like REQUEST packets, ACK packets are 6-byte packets, starting with the packet flag, followed by the opcode. Unlike REQUEST packets, the block number field of an ACK packet denotes the block number that is being acknowledged by the Pi. This ensures that the Pi correctly buffers the data in sequence. Figure 43 demonstrates the format of ACK packets.

An authentication module sends the token data to the Raspberry Pi using DATA packets. DATA packets are the only packets of variable length, with the DATA field ranging from 1 byte to 255 bytes long. Like ACK packets, the block number field denotes the block number of the data. The format of DATA packets is depicted Figure 44, but warrants further explanation. Figure 45 details the correct sequence of messages when an authentication module sends data to the Raspberry Pi. In this Figure, the NFC module is sending a 256-byte NFC authenticator.

As mentioned earlier in this section, the authentication module divides its buffer into 254-byte blocks, but the DATA packets can contain up to 255 bytes. This difference is due to the need to encode the data bytes before they are sent over USB. If the packet flag (0x7E) occurs in the data, then the packet will be cut short when

0	1 2	3 4	5	
FLAG	ACK	BLOCK	DATA	FLAG
0x7E	0x00 0x02		1 to 255 bytes	0x7E

Figure 44: Serial DATA packet format

it is received by the Raspberry Pi. To remedy this, the data block is encoded using the Consistent Overhead Byte Stuffing (COBS) algorithm.

COBS eliminates occurrences of the packet flag in the data by transforming them into pointers [58]. Appended to the start of the data is a byte, whose value denotes the index of the next occurrence of the packet flag, relative to that byte. If the next instance is not the terminal byte, then it will be transformed into a relative index of the next occurrence of the packet flag. This process of replacing the packet flag with relative indices is repeated until the terminal byte. Finally, the data is terminated with the packet flag. Table 1 shows a few sample inputs from SBACS hardware interactions, encoded using COBS.

Input Type	Input Bytes	COBS-Encoded Bytes
Module ID “nfc”	6E 66 63 00	04 6E 66 63 00 7E
PIN “0481”	30 34 38 31	04 30 34 28 31 7E
NFC data block	00 01 ... 7D 7E 7F ... FD	7F 00 01 ... 7D 80 7F ... FD 7E

Table 1: COBS encoding inputs and outputs for various data

This algorithm was chosen for its consistent behavior and low computational overhead [58]. COBS inserts one additional byte for every 254 bytes in a data block, requiring the data to be divided into 254-byte sections if it is larger than that. Therefore, to reduce the number of computations required to chunk the data for COBS and then for DATA packets, the DATA packet buffer size was chosen as 255 bytes to match the worst case for a COBS-encoded data chunk. This way, data that has been chunked for processing in COBS can be inserted into a packet without further transformations.

Before it is possible to initiate data transfers with authentication modules, the modules must first receive a SETUP packet from the Pi. This ensures that the modules will not attempt to gather authentication tokens before the core of the lock hardware system is ready. The SETUP packet is a 6-byte packet which does not use

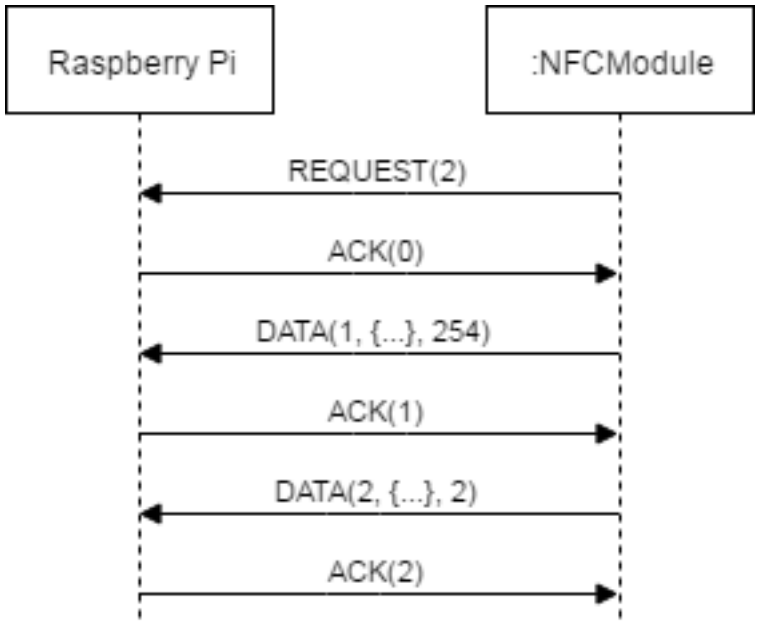

Figure 45: Hardware data transmission message sequence chart

0	1 2	3 4	5
FLAG	SETUP	BLOCK	FLAG
0x7E	0x11 0x11	x x	0x7E

Figure 46: Serial SETUP packet format

the block number field from the Serial Communication Protocol header, so those two bytes can be set to any arbitrary value. The format of SETUP packets is demonstrated in Figure 46.

Once a module has received a SETUP packet, it will respond with a DATA packet containing a string identifier for the module. Similarly to SETUP, the block number for this specific DATA packet does not matter and can be set to an arbitrary 2-byte value. The Raspberry Pi must acknowledge the identifier DATA packet with an ACK number matching what was provided. Figure 47 is a message sequence chart demonstrating the message sequence when the NFC module is set up.

The final type of packet defined in this protocol is the ERROR packet. These packets will be discussed in more detail in section 7.2.1, as their primary purpose is for debugging. The format of ERROR packets is given in Figure 48. Note that in

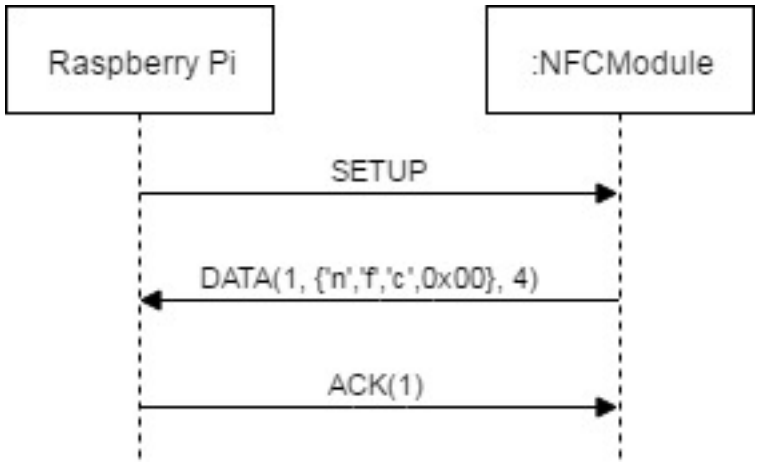

Figure 47: Hardware setup message sequence chart

0	1 2	3 4	5
FLAG	ERROR	CODE	FLAG
0x7E	0xFF 0xFF		0x7E

Figure 48: Serial ERROR packet format

the case of ERROR packets, the block number field from the packet header is used to specify an error code.

6.2.6 Video Stream

Implementing the video stream was simple when using a Raspberry Pi. After attaching a Raspberry Pi Camera Module to the camera input on the Pi, the video stream is started by typing the following command into a terminal:

```
raspivid -o - -t 0 -hf -w 640 -h 480 -fps 25 | cvlc
stream:///dev/stdin --sout '\#rtp{sdp=rtsp://:8554}'
:demux=h264
```

This command runs two programs. Firstly, it starts the "raspivid" program with the following settings:

- "-o -" specifies to output the video to the terminal,

- "-t 0" runs the stream with no timeout,
- "-hf" flips the image horizontally,
- "-w 640 -h 480 -fps 25" sets the video output to 640*480 resolution, at 25 frames per second.

The output of raspivid, the video stream, is then piped into another program, "cvlc". The cvlc program is used to convert the stream from raspivid into a RTP stream, viewable over the RTSP protocol. The "stream:///dev/stdin" parameter designates the terminal as the input for the stream. The "–sout '#rtpsdp=rtsp://:8554"" argument specifies that the stream output will be over RTSP, using port 8554. Finally, the video compression standard is designated as H.264 using the argument ":demux=h264"". H.264 was selected because it is one of the most commonly used formats for the compression and distribution of video content [59].

In order to avoid passing the threshold for the amount of "free" activity allowed on the Amazon Web Server, the video stream was streamed over LAN instead of through the server. This required a client to be aware of the lock's IP address, and the port being used for streaming. An implementation of the video stream that would have better fit a real-world scenario would have been to provide it only on client demand, and stream through the secure server.

6.3 Cloud Services

The server application was developed using Notepad++, a free source code editor with support for multiple languages [60]. This was used instead of an IDE to maintain the same development environment at school and outside of school when working on this project. Node.js does not require special build tools, so it is very easy to simply write the code in a relatively basic editor, and then run it from the command line to test it. To manage dependencies of libraries for our server code we used NPM, a package manager for Node.js. NPM is the dependency management tool that is bundled with Node.js, so it was a natural choice to use for our project [61].

6.3.1 Learning to Use Amazon Web Services

Amazon Web Services provides all the hosting that we needed for this project, so it was a natural choice for us to use (Section 5.5.1). When we started, we followed some basic tutorials to get a sample web application running, that would return a static HTML page on all HTTP requests. The first major challenge came from attempting to modify this sample application. First, we had to find the code of the sample application, as it was not given in any location that we thought to look on AWS. Eventually we found the code and modified it, then building and testing it locally to ensure that the code was correct. Then we had to deploy this new code, which proved to be harder than expected. The process is not complicated once it is known, but the wording was confusing in the instructions, so it took a few tries to get it working. Once we had this process figured out, we were able to easily write our own code and the process of updating our hosted server was easy.

The next challenge in learning AWS came from the database. Since we needed a persistent data storage, and had decided on a relational database, we decided to set up our database using AWS as well, which allowed us to keep all hosted services in one place. Initially, we created the database from the management console of our hosted web application. This made it easy to get a database setup and working with our server in very little time, but we soon realized that it was tied to the web application, and could not exist without the web application. This was not ideal, since the database should be able to exist separately, so we ended up deleting that database, and hosting one from the relational database manager on AWS. This kept it completely separate from the server, but still usable by the server. Another challenge was introduced when we tried to access the database directly, to initially build the tables and data, and to have the ability to query it later on. We were not all able to access the database, and we were unsure why, since we all were using the same tools, with the same settings to do it. Eventually we discovered that the database itself has settings to allow certain IP addresses to access it directly. We all agreed that this was a very smart feature, but we simply did not realize it existed at first. Once we discovered this, we were able to add the IP addresses of our computers as we needed to, but left the general public unable to access our database, keeping it secure.

We also had to learn more about how AWS handles SSL certificates and domain names, so that we could use HTTPS instead of HTTP to transfer user information securely. We realized we would need a domain name for certain features of our system, so we used AWS to purchase our domain name, and then to route requests to that domain name to our hosted server. This was not very complicated, but still had some learning to get it done correctly, and involved some change to configuration files on our server that we did not know existed up to that point. The specific troubles with SSL Certificates will be explained in more detail in the following section.

6.3.2 SSL Certificates

As discussed in Section 5.5.12, it is important to use HTTPS instead of HTTP when sending requests with sensitive information. In order to send requests over HTTPS instead of HTTP, SSL Certificates are needed. These provide an extra layer of security to ensure that only the sending and receiving parties are aware of the content of the message. SSL Certificates were a challenge to understand and work with, as none of us had much knowledge about them before. Certificates must be signed in order for them to be considered valid. This signing can be done by a developer, but the certificate must be explicitly declared as valid on each device that will have contact with the server over HTTPS in this case. This would be very hard to manage and is not practical for an application with many users. In order for most devices to know by default that it should be accepted, the certificates must be signed by a Certificate Authority. The Certificate Authorities are known to browsers and let the devices know that the destination can be trusted.

For our project it was decided that it would be easier for everyone if the certificates were signed by a Certificate Authority, to keep things simple. Unfortunately, this process can take time to get approved, and there is still some configuration to be managed by the server to work correctly. AWS handles many of these complications, and also handles the generation and signing of the certificates, but in order to do this, a domain name must be owned. Using AWS for this was the best option we found for this problem, so we bought a domain name, and we were able to get the

SSL Certificates, and had HTTPS working quickly after that.

6.3.3 Improving Response Time

The system needs to respond quickly to inputs from the user in order to provide the best user experience. One limiting factor of this response time is the communication with the cloud server. Sending a request over the network is very slow relative to other actions in the system, so the response from the server must be as quick as possible to minimize the delay. In order to keep this fast, the main goal of the server is to have a response sent as early as possible to the user. Many actions are dependent on one another, but in a few cases the response can be sent to the user before all the actions associated with the request have actually finished. For example, when a user logs in, the server must check that they have provided sufficient authentication to be logged in. Once this has been validated, the user is considered to be logged in. At this point, the server sends a key to the user that is valid for a limited time, which is used by the Android application to provide NFC authentication, and is also used in the authentication on subsequent requests to the server. After this key is sent back to the user, the server must still save this key if it is newly created, and must save the key as the NFC authenticator for the user. Since these actions can happen after the key is sent, the response will be returned faster.

Another limiting factor in the speed of server response is contact with the database. This can be slow, as the database may have a large amount of information to go through. To solve this problem, the server sends as few queries as possible, that will collect all the information that is needed for the request, but will not retrieve data that is unnecessary. Less queries to the database keeps the delay in sending and receiving responses low, allowing the server to spend less time waiting and more time processing information.

In addition, the calculations done to ensure security had to be measured to ensure that the response time was not suffering without meaningfully benefitting security. We measured the time taken by the password-based key derivation function using

the algorithm we chose, SHA256, given different numbers of iterations, which is summarised in table 2. We found that ten thousand iterations worked well, as roughly a tenth of a second did not dramatically impact user experience, while also ensuring that brute forcing was too inefficient. We then considered the salt length being used in the algorithm, but as table 3 shows, we found that the salt length did not have a meaningful effect on performance up to the suitably large length of 256 bytes.

Iterations	Response Time (ms)		
	Minimum	Maximum	Average
1	0.0383	1.382	0.0846
10	0.1566	1.9394	0.6024
100	1.2889	1.6009	1.4046
1000	11.0507	18.6680	14.0463
10000	110.9232	144.0115	119.2796
100000	1158.3401	1868.6968	1245.5582

Table 2: Time taken to perform SHA256 password based key derivation vs. iterations over 100 trials

Salt Length	Response Time (ms)		
	Minimum	Maximum	Average
1	111	151	121
2	111	160	122
4	111	151	123
8	111	148	122
16	110	151	124
32	110	150	122
64	111	156	123
128	110	175	124
256	111	152	123

Table 3: Time taken to perform SHA256 password based key derivation vs. salt length, with 10000 iterations over 100 trials

Chapter 7

Testing and Bug Fixes

7.1 Testing

7.1.1 Server Testing

To test the server, it was hosted locally by running the server code through the command line. In order to send HTTP requests to the server, an application called Postman was used. Postman is a useful application for building and sending HTTP requests, with a nice user interface [62]. It also saves the history of sent requests, making it easy to perform a suite of requests each time testing needs to be performed.

Once the code was tested locally and verified to be stable, it was deployed to Amazon Web Services to host the new version of the server. Once it was deployed, the test suite was run again to ensure that the code still worked in the actual server environment.

In order to debug when issues were found on the server, the server logs were very important. The logs of the server would often indicate how the input was interpreted or how the data from the database was read, so we could see where the issue was. In some cases, additional logging information was needed to indicate if certain blocks of code had been reached, to determine whether the program flow was correct.

7.1.2 Android Application Testing

The majority of the Android application's code was written for one of two reasons: to display information to the user, or to interface with the server controllers. While there exist ways to test the display [63], the display was not considered paramount to the success of the demonstrative system we set out to make. Conversely, the interactions that the application had with the controller are fundamental to the success of the project.

Since most of the interactions with the server a mock would have had to be built to set up local unit and regression tests. Since the server was in development at the same time as the application, there would have been extra work associated with keeping the mock up to date. Instead, most testing was done through a human regression over a test plan. The number of test cases to verify stayed at a reasonable level throughout the design process, so performing the regression by hand never became cumbersome. The regression covered each action that the application could take that interacted with the server including each unsuccessful interaction due to things like a dropped internet connection.

Bugs, when encountered, were examined using the debugger in Android Studio [47]. This debugger supports breakpoints as well as simple stepping instructions, which make finding the cause of an error much simpler. Another application of these features was in the regression tests themselves, where certain invisible values, such as the HMAC key received from the login controller on the server, could be checked without modifying the application.

7.1.3 Hardware Testing

Due to the hardware-driven nature of the lock control software, it was not possible to implement an automated method for testing the most of the software. The only classes that were hardware-independent were EncodedPacket, DecodedPacket, and AuthenticationToken. Some simple automated tests were written to ensure that data was correctly encoded when constructing an EncodedPacket, and correctly decoded when constructing a DecodedPacket. These tests were incorporated into the makefile,

and could be run by building the lock control software with the command "make test".

To test the lock control software's responses to serial data, an Arduino was programmed to send hard-coded data packets. For example, the Arduino could repeatedly transmit an empty SETUP packet. End-to-end tests could only be performed manually, but as there were only three cases to test, this was not an issue. The three end-to-end test cases were to test an authenticator that would result in access denied, test one authenticator which would grant access, and test two authenticators that combine to grant access.

7.2 Bugs

7.2.1 Issues with Serial Communication

Due to the limitations of the Arduino IDE, debugging the various components of the lock hardware was not a simple task. In particular, the largest limitation was that there was no way to control the execution of a program running on a microcontroller. One of the ways to remedy this was to use the Serial Monitor bundled with the IDE to monitor the data being sent over USB. However, it was not possible to run the Serial Monitor and the SBACS process at the same time, as only one of the processes would be allowed access to the serial port's device file. If the Arduino IDE Serial Monitor was running, the SBACS process could not. Since the Arduino IDE did not provide any other debugging mechanisms besides the Serial Monitor, the SBACS program would need to add the Serial Monitor's functionality to compensate for the device file issue.

This was accomplished by adding a "debug" option to the SBACS process code. The Debug class was added, which defined a DEBUG_LOG() function. This function would print messages to the command line, on the condition that a global variable "DEBUG" was defined. If the global variable was not defined, DEBUG_LOG() would do nothing. The DEBUG variable would be defined if the project was built using the "debug" option in the makefile.

This proved to be a more versatile method to debug than using the Serial Monitor,

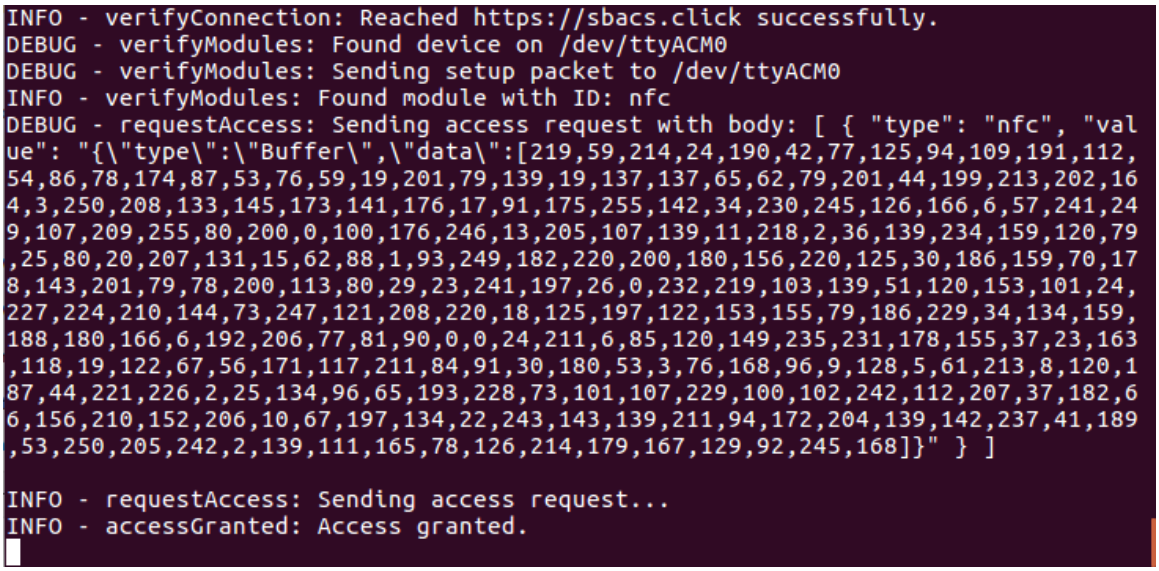

Figure 49: Hardware debug logging

as more information could be displayed, and in more formats. Bytes sent or received could be printed as hexadecimal arrays, as strings, or decimal arrays. Figure 49 shows the debug logging for a successful access attempt. The first four lines of the log demonstrate that the system starts up successfully, as it can reach the server, and locates at least one authentication module. The access attempt was initiated between the fourth and fifth lines of the log; note that it was not necessary to print every byte sent and received by the SBACS process, as all serial communication issues had been resolved by the end of the project.

In addition to the Debug class, the ERROR packet from the Serial Communication Protocol was used by the authentication modules to notify the core process of any issues encountered, either with the module itself, or when receiving serial data. The three major errors that the authentication modules could encounter were issues during setup, receiving a packet with an incorrect opcode, and errors decoding the received packet. Setup errors would be due to an issue with the authentication module, such as a wire being disconnected, preventing the module from setting up properly. Opcode and decoding errors were indicative of a problem with the serial connection between the module and Raspberry Pi. One case where this occurred was due to a faulty USB

cable, while in another it was due to the Raspberry Pi's serial port being incorrectly configured.

7.2.2 Issues with String Encodings

When the database was first built, we decided to store authentication data as strings. This was chosen because we found strings easy to work with, and most HTTP libraries had support for sending data as strings. During the integration phase of the project, we started having trouble with authentication, where the data that was retrieved from the server by the phone was then sent to the lock controller and then sent back to the server for validation. The server was receiving different data than it was initially sending, so all requests to unlock were being denied. After debugging at different points in the system, we came to the conclusion that the encoding of strings was not the same across the languages and technologies we were using, so the data was not the same when it reached the server again.

To solve this problem, we changed the data types stored by the database to be binary data rather than strings. This eliminated the problem of encoding, since binary data is simply binary regardless of the platform or technology being used. We realized that it would have made more sense to go with a binary data type in the beginning, but we did not anticipate the problems with encoding initially. This problem took a long time to identify, and was the largest delay in our integration phase of the project.

7.2.3 Issues with Encryption Algorithm Availability

This issue is discussed briefly in section 6.1.2. The algorithm used by the server to compute HMAC message digests used the password-based key derivation function (PBKF2) [53] run using SHA256 [51]. We chose SHA256 over SHA1 because SHA1 has a known method of generating collisions [64]. However, Android did not specifically state that PBKF2 over SHA256 was supported on all devices [65]. This unfortunately caused problems later on in development. After a certain update, the Android devices that we were using for testing stopped supporting this algorithm.

We had a few options: we could change to the supported SHA1 algorithm, we

could make use of a third party implementation of the algorithm using SHA256, we could write the SHA256 implementation ourselves, or we could change the method used to create message digests. Each of these options had benefits and drawbacks, but the decision had to be made quickly since the change to support occurred very near to a deadline.

Using a third party implementation seemed risky without the time to investigate the various options to be sure that the code met the standards needed. There was also a possibility of difficulties integrating the code with ours, and potential legal problems with licensing. While this may have required the least code change, we felt the risk associated with this option was too great.

Implementing the algorithm ourselves would almost certainly have not been feasible within the time required. Further, writing such complicated and important algorithms is generally discouraged, as even a small error could result in incompatibility with other code using the standards correctly, or even worse, a security breach. Overall, this option was not appealing.

Eliminating those options left either changing a couple constants in the code to switch the underlying secure hash algorithm, or changing the way keys are generated to using a different supported algorithm. For the time being, we decided that the much simpler option of converting the code to use SHA1 instead of SHA256. Since then, we have updated the server and application to make use of the built in HMAC digest generation code using the currently collision-free SHA256.

Chapter 8

Conclusions

The SBACS provides a fairly simple method of digitising any physical lock. The digital locks are secure, reusable, and easier to recover than physical locks are. The current implementation of the system can be used to maintain a simple business use case, such as the online order secure pickup use case described in section 4.1. Locks and users can be added to the system, and registrations can be made between them. Then, the users can use their identities to authenticate themselves and open the locks.

The system could be made more accessible to companies that don't want to use NFC or simple PIN codes as their authentication method by implementing more authentication modules. Certain companies would want a more secure method of authentication, which would encourage developing a module that performs facial, fingerprint, or other biometric recognition. The SBACS's design ensures that the only difficult work would be in designing the software to actually compare the biometrics. Integration of the new modules is totally invisible to the services provided.

In addition, several companies would want to be able to associate further information with registrations, beyond just the video streaming, such as an inventory of items or a list of accesses to the lock. These could be implemented with minimal changes to the core of the SBACS. The authentication could be handled by the same session and HMAC methods used elsewhere, so the only real change would be an additional column or table, depending on the complexity of the desired data. The majority of the work would reside in providing an interface for collecting the data.

Currently, the SBACS is an entirely flat system with respect to privileges for users. In a practical SBACS having a distinction for administrative users would likely be beneficial. This would require work on the server to support, particularly if multiple levels of administrator were added. Multiple types of administrators would make sense for providing SBACS as a service.

A large number of the difficulties that came from integration came from the complicated hardware logic. If the SBACS were to be continued, likely a good decision would be to offload as much work as possible from the hardware. For example, we could have used a simple card reader instead of an NFC shield. This would have required the application to communicate with the server after reading the lock identity from the card, and would require either that the server remember each lock, or for the card reader to store the IP address as well. This would have complicated the application and the server, but the gain in simplifying the hardware would outweigh those costs.

An alternative solution that has been discovered recently by the group is continuous integration and continuous delivery for the cloud server component of the system. This technique was not used, as we did not know enough about it, but could have been used to quickly perform more testing and automatically deploy code that works to the server. Continuous integration would allow our server code to be tested automatically every time we commit a new version to our version control system. In the case where all the tests pass, the code could then be automatically deployed to the server using continuous deployment. These two techniques used together could allow for faster discovery of bugs, and a faster process of deployment to the server.

Bibliography

[1] Board of Governors of the Federal Reserve System. *Consumers and Mobile Financial Services*. 2015. URL: http://www.federalreserve.gov/econresdata/consumers-and-mobile-financial-services-report-201503.pdf (visited on 09/15/2016).

[2] Walmart. *Grab And Go*. 2016. URL: http://www.walmart.ca/en/help/checkout/grab-and-go (visited on 08/19/2016).

[3] Tony R. Kuphaldt. *Chapter 3 - Electrical Safety*. URL: https://www.allaboutcircuits.com/textbook/direct-current/chpt-3/ohms-law-again/ (visited on 03/07/2017).

[4] Github. *Github*. URL: https://github.com/ (visited on 04/06/2017).

[5] NearFieldCommunications.org. *About Near Field Communication*. 2016. URL: http://nearfieldcommunication.org/about-nfc.html (visited on 09/08/2016).

[6] NFC Forum. *What is NFC? About the Technology*. 2016. URL: http://nfc-forum.org/what-is-nfc/about-the-technology/ (visited on 09/11/2016).

[7] qrscanner.us. *Near Field Communication Payment and NFC Payment*. URL: http://www.qrscanner.us/nfc-payment.html (visited on 09/11/2016).

[8] Ernst Haselsteiner and Klemens Breitfu. *Security in Near Field Communication (NFC)*. 2006. URL: http://events.iaik.tugraz.at/RFIDSec06/Program/papers/002%20-%20Security%20in%20NFC.pdf (visited on 09/11/2016).

[9] Amazon. *What is Cloud Computing? - Amazon Web Services*. URL: https://aws.amazon.com/what-is-cloud-computing/ (visited on 04/01/2017).

[10] J. F. Kurose and K. W. Ross. *Computer Networking. A Top-Down Approach Featuring the Internet*. 7th ed. Pearson, 2017.

[11] R. Cellan-Jones. "A 15 pound computer to inspire young programmers". In: *BBC News* (May 5, 2011). URL: http://www.bbc.co.uk/blogs/thereporters/rorycellanjones/2011/05/a_15_computer_to_inspire_young.html (visited on 04/03/2017).

[12] Samuel Gibbs. "Raspberry Pi becomes best selling British computer". In: *The Guardian* (Feb. 18, 2015). URL: https://www.theguardian.com/technology/2015/feb/18/raspberry-pi-becomes-best-selling-british-computer (visited on 04/01/2017).

[13] Raspberry Pi Foundation. *Raspberry Pi 3 Model B*. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (visited on 04/03/2017).

[14] SparkFun Electronics. *What is an Arduino?* URL: https://learn.sparkfun.com/tutorials/what-is-an-arduino (visited on 04/05/2017).

[15] Xeltek. *Microcontroller Programming*. 2017. URL: http://www.xeltek.com/resources/technical-articles/memory-device-types/microcontroller-programming/ (visited on 04/05/2017).

[16] Madeline Farber. *Consumers Are Now Doing Most of Their Shopping Online*. Fortune. July 8, 2016. URL: http://fortune.com/2016/06/08/online-shopping-increases/ (visited on 04/04/2017).

[17] Microsoft Developer Network. *Chapter 3: Architectural Patterns and Styles*. URL: https://msdn.microsoft.com/en-ca/library/ee658117.aspx#ClientServerStyle (visited on 04/01/2017).

[18] Mark Richards. *Software Architecture Patterns*. 1st ed. O'Reilly Media, Inc, 2015.

[19] ISO/IEC JTC 1/SC 17, ed. *Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange*. International Organization for Standardization, Geneva, Switzerland., Apr. 2013. URL: https://www.iso.org/standard/54550.html (visited on 03/27/2017).

[20] Gartner. *Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016*. URL: http://www.gartner.com/newsroom/id/3609817 (visited on 03/27/2017).

[21] Buster Hein. *Apple confirms iPhone 6 NFC chip is only for Apple Pay at launch*. Cult of Mac. URL: http://www.cultofmac.com/296093/apple-confirms-iphone-6-nfc-apple-pay/ (visited on 03/27/2017).

[22] Android Developers. *Host-based Card Emulation*. URL: https://developer.android.com/guide/topics/connectivity/nfc/hce.html (visited on 03/26/2017).

[23] Android Developers. *Near Field Communication*. URL: https://developer.android.com/guide/topics/connectivity/nfc/index.html (visited on 03/27/2017).

[24] ISO/IET JTC 1/SC 17, ed. *Identification cards – Integrated circuit cards – Part 5: Registration of application providers*. International Organization for Standardization, Geneva, Switzerland., Dec. 2004. URL: https://www.iso.org/standard/34259.html (visited on 03/27/2017).

[25] Android Developers. *Implementing an HCE Service*. URL: https://developer.android.com/guide/topics/connectivity/nfc/hce.html#ImplementingService (visited on 04/01/2017).

[26] ISO/IEC JTC 1/SC 17, ed. *Identification cards – Contactless integrated circuit cards – Proximity cards – Part 4: Transmission protocol*. International Organization for Standardization, Geneva, Switzerland., July 2008. URL: https://www.iso.org/standard/50648.html (visited on 03/27/2017).

[27] Glenn E. Krasner and Stephen T. Pope. "A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80". In: *J. Object Oriented Program.* 1.3 (Aug. 1988), pp. 26–49. ISSN: 0896-8438. URL: http://dl.acm.org/citation.cfm?id=50757.50759.

[28] Android Developers. *Activites*. URL: https://developer.android.com/guide/components/activities/index.html (visited on 04/06/2017).

[29] Android Developers. *Intent*. URL: https://developer.android.com/reference/android/content/Intent.html (visited on 04/01/2017).

[30] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[31] Premier Farnell Group. *Arduino Uno*. URL: http://www.farnell.com/datasheets/1682209.pdf (visited on 04/06/2017).

[32] Amazon. *What is AWS? - Amazon Web Services*. URL: https://aws.amazon.com/what-is-aws/ (visited on 03/09/2017).

[33] Roy T. Fielding. *Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)*. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (visited on 03/09/2017).

[34] ECMA International. *Final draft of the TC39 The JSON Data Interchange Format standard*. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (visited on 03/09/2017).

[35] W3 Schools. *Introduction to HTML*. URL: https://www.w3schools.com/html/html_intro.asp (visited on 03/26/2017).

[36] jQuery Foundation. *Browser Support - jQuery*. URL: http://jquery.com/browser-support/ (visited on 03/26/2017).

[37] jQuery Foundation. *jQuery*. URL: http://jquery.com/ (visited on 03/26/2017).

[38] Mike Wasson. *ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET*. URL: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx (visited on 11/30/2013).

[39] TutorialsPoint. *SQL RDBMS Concepts*. URL: https://www.tutorialspoint.com/sql/sql-rdbms-concepts.htm (visited on 04/01/2017).

[40] TutorialsPoint. *Database - First Normal Form (1NF)*. URL: https://www.tutorialspoint.com/sql/first-normal-form.htm (visited on 04/01/2017).

[41] TutorialsPoint. *Database - Second Normal Form (2NF)*. URL: https://www.tutorialspoint.com/sql/second-normal-form.htm (visited on 04/01/2017).

[42] TutorialsPoint. *Database - Third Normal Form (3NF)*. URL: https://www.tutorialspoint.com/sql/third-normal-form.htm (visited on 04/01/2017).

[43] Defuse Security. *Salted Password Hashing - Doing it Right*. URL: https://crackstation.net/hashing-security.htm#salt (visited on 09/26/2016).

[44] TutorialsPoint. *HTTP Tutorial*. URL: https://www.tutorialspoint.com/http/ (visited on 03/26/2017).

[45] Tanvi Vyas Peter Dolanjski. *Communicating the Dangers of Non-Secure HTTP*. URL: https://blog.mozilla.org/security/2017/01/20/communicating-the-dangers-of-non-secure-http/ (visited on 01/20/2017).

[46] Martin Fowler. *P of EAA: Front Controller*. URL: https://martinfowler.com/eaaCatalog/frontController.html (visited on 04/01/2017).

[47] Android Developers. *Android Studio*. URL: https://developer.android.com/studio/index.html?gclid=CPmEm9SPhNMCFRKewAodWRYK0w (visited on 04/01/2017).

[48] Gradle. *Gradle Build Tool*. URL: https://gradle.org/ (visited on 04/01/2017).

[49] Android Developers. *Transmitting Network Data Using Volley*. URL: https://developer.android.com/training/volley/index.html (visited on 04/01/2017).

[50] Google. *Volley*. URL: https://github.com/google/volley (visited on 04/01/2017).

[51] D. Eastlake 3rd and T. Hansen. *US Secure Hash Algorithms (SHA and HMAC-SHA)*. RFC 4634. RFC Editor, 2006, pp. 1–108. URL: http://www.rfc-editor.org/info/rfc4634.

[52] K. Ed. Moriarty, B. Kaliski, and A. Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. RFC 8018. RFC Editor, 2017, pp. 1–40. URL: http://www.rfc-editor.org/info/rfc8018.

[53] S. Josefsson. *PKCS #5: Password-Based Key Derivation Function 2 (PBKDF2) Test Vectors*. RFC 6070. RFC Editor, 2011, pp. 1–5. URL: http://www.rfc-editor.org/info/rfc6070.

[54] K. Whites. *Lecture 4: RL Circuits. Inductive Kick. Diode Snubbers.* South Dakota School of Mines and Technology. 2017. URL: http://whites.sdsmt.edu/classes/ee322/class_notes/322Lecture4.pdf (visited on 04/06/2017).

[55] ON Semiconductor. *Plastic Medium-Power Complementary Silicon Transistors*. 2014. URL: http://www.onsemi.com/pub/Collateral/TIP120-D.PDF (visited on 03/26/2017).

[56] LC Circuits. *Relay Module*. URL: http://elecfreaks.com/store/download/datasheet/breakout/Relay/Relay.pdf (visited on 10/19/2016).

[57] K. Sollins. *The TFTP Protocol (Revision 2)*. RFC 1350. RFC Editor, 1992, pp. 1–11. URL: http://www.rfc-editor.org/info/rfc1350.

[58] S. Cheshire and M. Baker. "Consistent Overhead Byte Stuffing". In: *Association for Computing Machinery SIGCOMM '97*. 1997. URL: http://conferences.sigcomm.org/sigcomm/1997/papers/p062.pdf (visited on 04/04/2017).

[59] J. Ozer. *Encoding for Multiple Screen Delivery, Section 3, Lecture 7: Introduction to H.264*. URL: https://www.udemy.com/encoding-for-multiple-screen-delivery/learn/v4/t/lecture/2014750 (visited on 04/05/2017).

[60] Don Ho. *Notepad++*. URL: https://notepad-plus-plus.org/ (visited on 04/01/2017).

[61] Isaac Schlueter. *Build amazing things*. URL: https://www.npmjs.com/ (visited on 04/01/2017).

[62] Inc. Postdot Technologies. *Postman*. URL: https://www.getpostman.com/ (visited on 04/01/2017).

[63] Android Developers. *Automating User Interface Tests*. URL: https://developer.android.com/training/testing/ui-testing/index.html (visited on 04/04/2017).

[64] Google. *Announcing the first SHA1 collision*. 2017. URL: `https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html` (visited on 04/04/2017).

[65] Android Developers. *SecretKeyFactory*. URL: `https://developer.android.com/reference/javax/crypto/SecretKeyFactory.html` (visited on 04/04/2017).

Appendix A

Diagrams

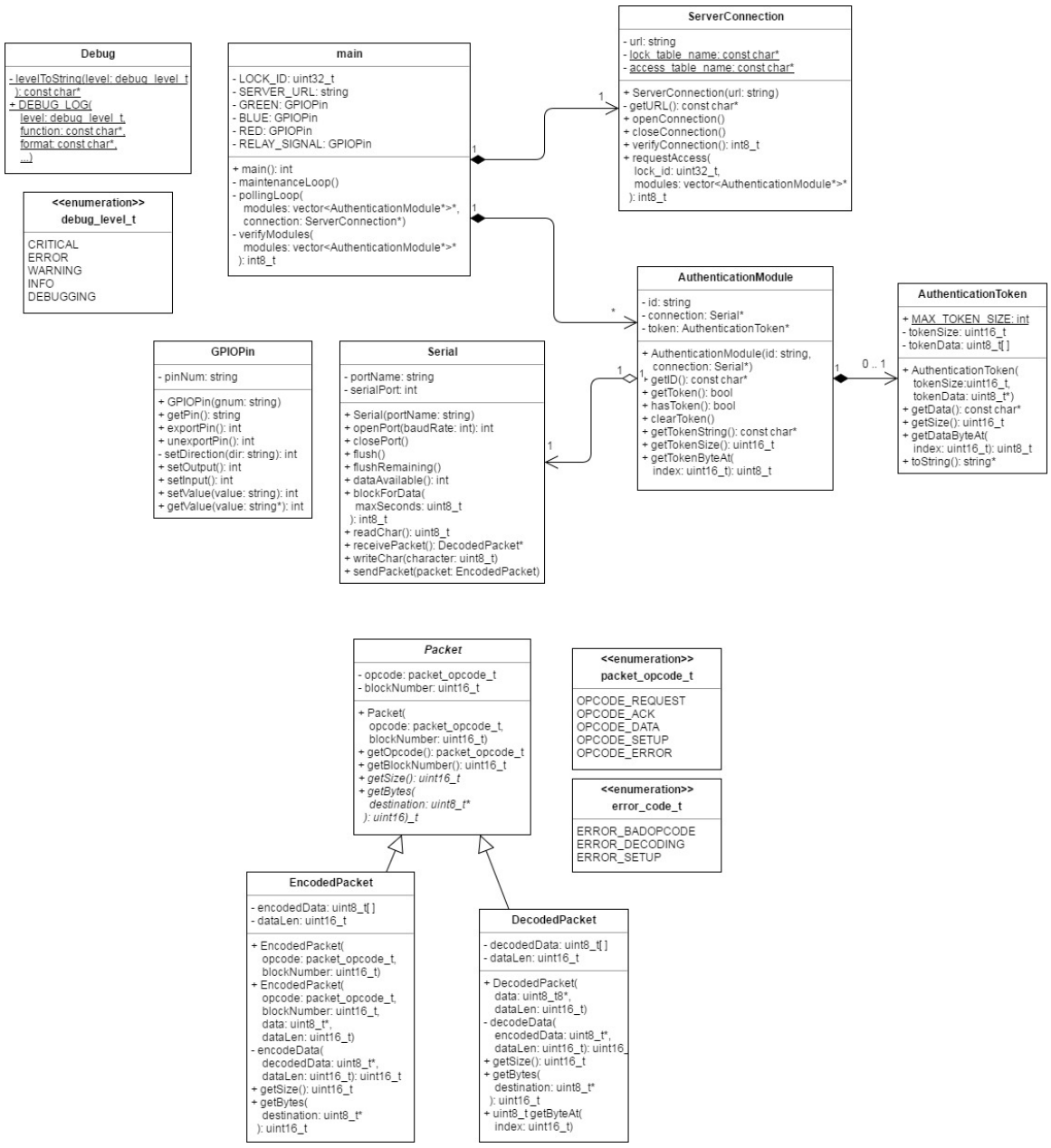

Figure 50: Lock control software full class diagram