**aMAZE-RT: Secure Management for OpenWRT**

|  |
| --- |
| A Project Report Presented to  The Faculty of the College of Engineering |
| San Jose State University In Partial Fulfillment Of the Requirements for the Degree  **Master of Science in Software Engineering** |
| By |
| Ginto George, Binu Jose, Sandeep Panakkal, Nabin Thomas |
| December/2020 |

|  |
| --- |
| Copyright © 2020 |
| Ginto George, Binu Jose, Sandeep Panakkal, Nabin Thomas |
| ALL RIGHTS RESERVED |

|  |
| --- |
| **APPROVED** |
|  |
| Prof. Kaikai Liu, Project Advisor |
|  |
| [Program Director’s Name], Director, MS Software Engineering |
|  |
| [Department Chair’s Name], Department Chair |

ABSTRACT

aMAZE-RT: Secure Management for OpenWRT

By Ginto George, Binu Jose, Sandeep Panakkal, Nabin Thomas

A router is a device that helps to interconnect two or more networks, by forwarding data packets between them [1]. Home and Small office networks typically connect to the internet via consumer-level Wi-Fi routers, which provide wireless access points to the devices at home and route packets to the modem.

Typical consumer routers have locked firmware which restricts the functionality to whatever the device manufacturer decides to include. Open Source software stack like OpenWRT (Open Wireless Router) can be used to enhance the functionality and device security of consumer and custom Wi-Fi routers. Management of OpenWRT based devices is done using a browser or command-line interface. While this may work for the tech-savvy, a large majority of users may not be comfortable managing their devices this way. Some consumer-level devices have moved towards mobile app-based management, but OpenWRT lacks a mobile app-based router management functionality.

In this project, we plan to develop an end-to-end secure software to manage routers using a mobile application, with a customized version of OpenWRT, a mobile app as the front end, and an optional cloud-based management system for communication. The app provides features like monitoring and managing connection attempts with the notification on the mobile app via the cloud. Mobile App shall manage router connection attempts as an additional authentication factor on top of the Wi-Fi user authentication, enabling Multi-Factor Authentication. We also plan to port the major features of the LuCI web-based management tool for OpenWRT to the App, such as Access Control, package management, real-time monitoring, etc.

|  |
| --- |
| **Acknowledgments** |
| The authors are deeply indebted to Professor Michael Tjebban and Advisor Professor Kaikai Liu for their invaluable comments and assistance in the preparation for this project. |

**Table of Contents**

[Chapter 1. Project Overview 1](#_Toc56617560)

[Introduction 1](#_Toc56617561)

[Proposed Areas of Study and Academic Contribution 3](#_Toc56617562)

[Current State of the Art 4](#_Toc56617563)

[State-of-the-Art Summary 6](#_Toc56617564)

[Project Justification 7](#_Toc56617565)

[Chapter 2. Project Architecture 8](#_Toc56617566)

[Introduction 8](#_Toc56617567)

[Architecture Subsystems 8](#_Toc56617568)

[Chapter 3. Technology Descriptions 10](#_Toc56617569)

[Client Technologies 10](#_Toc56617570)

[Mobile App 10](#_Toc56617571)

[I. Android SDK 10](#_Toc56617572)

[II. JSch 10](#_Toc56617573)

[III. GSon 11](#_Toc56617574)

[IV. Firebase Cloud Messaging 11](#_Toc56617575)

[AmazeRT Agent 11](#_Toc56617576)

[I. Python with Websockets 12](#_Toc56617577)

[II. UUID 12](#_Toc56617578)

[Middle-Tier Technologies 13](#_Toc56617579)

[App Engine 13](#_Toc56617580)

[Cloud Function 14](#_Toc56617581)

[Websockets with TLS for security for device to App Engine communication 14](#_Toc56617582)

[Data-Tier Technologies 14](#_Toc56617583)

[Secure Data Storage 14](#_Toc56617584)

[FireBase RealTime Database 14](#_Toc56617585)

[File Encryption for App data storage 15](#_Toc56617586)

[Secure Device to Mobile App data transfer 15](#_Toc56617587)

[Chapter 4. Project Design 16](#_Toc56617588)

[Mobile App 16](#_Toc56617589)

[App Login UI 16](#_Toc56617590)

[Device Registration UI 16](#_Toc56617591)

[Device Settings UI 18](#_Toc56617592)

[I. Settings fetch 18](#_Toc56617593)

[II. Settings push 19](#_Toc56617594)

[Device Status UI 19](#_Toc56617595)

[Securing data in transit 19](#_Toc56617596)

[User Profile UI 20](#_Toc56617597)

[AmazeRT Agent 20](#_Toc56617598)

[Registration and Installation 21](#_Toc56617599)

[Cloud Registration 21](#_Toc56617600)

[Heartbeats and Status Updates 22](#_Toc56617601)

[Device Configuration Management 23](#_Toc56617602)

[Failure Handling 23](#_Toc56617603)

[Cloud Backend 24](#_Toc56617604)

[Websocket 24](#_Toc56617605)

[AppEngine 24](#_Toc56617606)

[Cloud function 24](#_Toc56617607)

[Secure Data Storage 24](#_Toc56617608)

[Shared Configuration Database 24](#_Toc56617609)

[Data Access Control 24](#_Toc56617610)

[Device Registration data on Mobile app 25](#_Toc56617611)

[Chapter 5. Project Implementation 26](#_Toc56617612)

[Mobile App 26](#_Toc56617613)

[App Login UI 26](#_Toc56617614)

[Device Registration UI 26](#_Toc56617615)

[SymKeyEncryption class 28](#_Toc56617616)

[FirebaseQueryLiveData class 29](#_Toc56617617)

[Device Settings UI 29](#_Toc56617618)

[Device Status UI 30](#_Toc56617619)

[User Profile UI 30](#_Toc56617620)

[AmazeRT Agent 30](#_Toc56617621)

[Registration and Installation 31](#_Toc56617622)

[Cloud Registration 32](#_Toc56617623)

[Status Updates and Heartbeats 35](#_Toc56617624)

[Device Configuration Management 36](#_Toc56617625)

[Failure Handling 37](#_Toc56617626)

[Cloud Backend 39](#_Toc56617627)

[Websocket 39](#_Toc56617628)

[AppEngine 39](#_Toc56617629)

[Cloud functions 39](#_Toc56617630)

[Secure Data Storage 40](#_Toc56617631)

[Shared Configuration Database 40](#_Toc56617632)

[Device Registration data on Mobile app 40](#_Toc56617633)

[Hardware platform 40](#_Toc56617634)

[Chapter 6. Testing and Verification 41](#_Toc56617635)

[Testing Process 41](#_Toc56617636)

[Mobile App 41](#_Toc56617637)

[I. Database rules validation 41](#_Toc56617638)

[II. UI 41](#_Toc56617639)

[AmazeRT Agent 42](#_Toc56617640)

[I. Installation 42](#_Toc56617641)

[II. Initial configuration 42](#_Toc56617642)

[III. Cloud Registration 42](#_Toc56617643)

[IV. Device Configuration Changes 43](#_Toc56617644)

[V. Failure Recovery 43](#_Toc56617645)

[Cloud Backend 44](#_Toc56617646)

[Database testing 44](#_Toc56617647)

[Test results 44](#_Toc56617648)

[Chapter 7. Performance and Benchmarks 45](#_Toc56617649)

[Scalability 45](#_Toc56617650)

[Throughput 45](#_Toc56617651)

[Reliability 45](#_Toc56617652)

[Feature Parity 45](#_Toc56617653)

[Chapter 8. Deployment, Operations, Maintenance 46](#_Toc56617654)

[Cloud Backend 46](#_Toc56617655)

[Cloud Functions 46](#_Toc56617656)

[App Engine 46](#_Toc56617657)

[Android App and Device software 46](#_Toc56617658)

[Android App Stores 46](#_Toc56617659)

[Packaging device software 46](#_Toc56617660)

[Secure Database operations 46](#_Toc56617661)

[Firebase Realtime Database 46](#_Toc56617662)

[Local App database 46](#_Toc56617663)

[Device data storage 46](#_Toc56617664)

[Github repository 47](#_Toc56617665)

[Chapter 9. Summary, Conclusions, and Recommendations 48](#_Toc56617666)

[Summary 48](#_Toc56617667)

[Conclusions 48](#_Toc56617668)

[Recommendations for Further Research 48](#_Toc56617669)

[Glossary 50](#_Toc56617670)

[References 51](#_Toc56617671)

[Appendices 52](#_Toc56617672)

**List of Figures**

[Figure 1 Typical Configuration of Home Network 1](#_Toc56621954)

[Figure 2 High Level Components 3](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621955)

[Figure 3 LuCL Web App UI – 1 5](#_Toc56621956)

[Figure 4 LuCL Web App UI – 2 5](#_Toc56621957)

[Figure 5 LuCL Web App UI - 3 6](#_Toc56621958)

[Figure 6 Project Architecture 8](#_Toc56621959)

[Figure 7 Cloud Architecture 13](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621960)

[Figure 8 App Launch page 17](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621961)

[Figure 9 App Federated Login 18](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621962)

[Figure 10 Device List screen with fab button to add device 19](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621963)

[Figure 11 App, Add device for registration 20](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621964)

[Figure 12 App, Device Registration Progress 21](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621965)

[Figure 13 App, Device registration Done 22](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621966)

[Figure 14 App, Device List with newly added device 23](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621967)

[Figure 15 Device Registration Sequence Diagram 25](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621968)

[Figure 16 App User Profile screen 28](file:///C:\Users\gintgeor\AndroidStudioProjects\amazert\Learn\PrjReport\CMPE_295B_Final_Project_Report.docx#_Toc56621969)

**List of Tables**

[Table 1 Glossary 22](#_Toc55386090)

# Project Overview

## Introduction

* + 1. A router is a device that interconnects two or more networks, by forwarding data packets between them [1]. Routers can interconnect multiple public networks on the internet, such as the ones used in the internet backbone and interconnections between internet service providers [1]. Internet access in small offices and home environments typically is done via a broadband modem that connects to the public internet via the Internet Service provider. On the internal network, multiple devices may share the internet connection by using a wireless router, which interconnects the local network to the public network via the broadband modem. Figure 1 - Typical Configuration of Home Network shows how a typical home network connects to the internet using a wireless router.
  1. **Diagram

     Description automatically generated**

Figure Typical Configuration of Home Network

The consumer router market comes with a plethora of brands and models supporting varying hardware and software features. They often restrict the user with locked firmware limiting the features and functionality to a subset of possible features, as conceived by the device manufacturer. Like in the case of many commercial software products, Open Source software stacks provide an alternate option to enhance the functionality and security of consumer routers. OpenWRT is such an Open Source Linux based Wireless router firmware. OpenWRT supports a wide range of hardware. There are many commercial products available based on OpenWRT. Customers can add packages to enhance the functionality of the router. OpenWRT devices provide web-based administration that provides very advanced configuration and management functionality. This would not work for a large majority of novice users who would find it hard and technically demanding to manage their devices this way. There are only a handful of consumer-oriented devices that have ventured towards mobile application-based device management. This is one area that OpenWRT falls behind and even a basic mobile application-based router management functionality will benefit all the consumers in a big way. The trend towards using mobile apps for everything has made routers with mobile App based management more appealing for a large group of users, who would otherwise not even bother to configure the router with the web-based management tools [2].

As part of this project, following components are implemented that together can accomplish this objective:

1. A new OpenWRT package to support the core device handling logic.

2. A Mobile App for the end user to control and interact with his OpenWRT device.

3. The Cloud component to facilitate the Mobile App to OpenWRT device interactions.

Figure 2 gives a high-level view of the components

Figure High Level Components

The Mobile App is envisioned as the single point for the OpenWRT device management. Once the OpenWRT device is out of the box, connected and powered up, the Mobile App takes over. It allows the user to perform the initial device setup and registration with the amazeRt cloud infrastructure. The initial setup takes care of:

1. Authenticating this first-time user as the admin user, using the secure Auth APIs provided by Google Firebase Auth infrastructure.

2. Installing the packaged amazeRt software on to the customer device being setup.

3. Setting up the Symmetric Key used for securing the communication between this OpenWRT device and its management App

4. Registering the device and the user with the amazeRt cloud infrastructure.

Once the initial setup is completed, the Mobile App provides the authenticated user access to the various OpenWRT device settings and live Status updates. A prominent feature in development now is the ability for the Authenticated Mobile App user to block a malicious device that is connected to the OpenWRT device.

## Proposed Areas of Study and Academic Contribution

Need to add relevant information here.

## Current State of the Art

OpenWRT firmware is based on the Linux operating system. The system provides a shell (ash shell) for running commands for making configuration changes. The fact that this is based on Linux makes it easy for extending the functionality, unlike other firmware options. OpenWRT provides a standardized way of implementing enhancements, called packages [3]. Each package is analogous to an application that can be installed on the device, extending its functionality. OpenWRT community also develops and maintains a list of around 3500 packages [4]. A Web-based management interface is provided which can be installed as a package on the device [2]. like art is currently supported in a wide range of router hardware. OpenWRT provides LuCI Web-based user interface for router administration and monitoring [5]. The UI is a bit complex to manage and operate for most of the home consumer router users. The following figures give a glimpse of the complex UI from OpenWRT

* 1. Graphical user interface

     Description automatically generated

Figure LuCL Web App UI – 1

* 1. A picture containing graphical user interface

     Description automatically generated

Figure LuCL Web App UI – 2

* 1. Graphical user interface, application

     Description automatically generated

Figure LuCL Web App UI - 3

For the not so tech-savvy end-user, management via a Web-based UI can be an uninteresting task when it involves remembering the IP Address, requiring login every time an administration or monitoring task needs to be done. This also has a negative impact on the security aspect of the router, since the users are mostly unknown about who is connected to their network, or what apps are using most data. There exist some hobby projects that provide basic mobile app functionality, but they are limited to basic management functions, and do not have any cloud support.

### State-of-the-Art Summary

There is no doubt that OpenWRT is an excellent open-source alternative for router firmware, providing one of the richest sets of management and security tools. However, having only a command line or web-based management and monitoring interface is a deterrent for most consumers in the era of mobile apps.

## Project Justification

OpenWRT is a feature-rich open-source wireless router firmware. It has a rich set of features and together with the active community support and contribution makes it a leader in its arena. The OpenWRT software stack is well adopted by a wide range of hardware vendors. Despite the rich feature set and much-desired hardware adoption, there is one much-desired improvement to the software stack, a mobile management app. When it comes to management and monitoring interfaces, the lack of a mobile app for continuous monitoring and configuration management is the Achilles heel of the system. No matter how advanced the available web interface features are, their usage mandates a browser and logging in every time something needs to be checked. Mobile app-based configuration and monitoring can solve this drawback by providing a simple user-friendly management and monitoring interface. In this era of mobile apps for everything, the availability of an average mobile application-based interface would cater to most of the internet users of the world. OpenWRT lacks in this area with no serious mobile application-based management and monitoring solutions being developed.

The project aims to improve the user adaptability and adoption rate of the OpenWRT stack. This will be done by adding a mobile application-based interface to manage and monitor the OpenWRT router. Cloud-based support for real-time monitoring and management of the router will be added to further enhance the adoption rate and user-friendliness of the software stack. The result will be an OpenWRT based software stack that supports mobile app-based management and monitoring of OpenWRT routers.

# Project Architecture

## Introduction

Include introductory text plus a diagram.

Diagram

Description automatically generated

Figure Project Architecture

TODO: Redo the Diagram as vector

## Architecture Subsystems

The entire project is divided into 3 major architectural subsystems as below.

1. The Mobile App used to provide the front-end functionality
2. The software that runs on the OpenWRT providing status updates and handling configuration changes
3. The business logic to manage and handle communication between OpenWRT device and Mobile App, which runs on Google Cloud Platform
4. The Secure Shared Database which keeps track of the device settings and provides real-time notifications to Mobile App about configuration changes

# Technology Descriptions

Assume you audience is a skilled computer scientist that has some familiarity with technologies taught in the client/server program. The topics below are for a typical MS Software Engineering project. Adjust the topics in this chapter to meet the needs of your project.

## Client Technologies

### Mobile App

Mobile application is an Android App on the mobile device running Android OS level 29 or later. This is a front-end application to configure and control the OpenWRT device. The functionalities include device registration, device configuration, device status update, notification handling, device removal.

1. Android SDK

Android SDK is a comprehensive set of development tools for Android app development. The tools include libraries, debugger, QEMU based handset emulator, tutorials, sample codes and documentation. Android application is packaged in “.apk” format. The apk package contains Dalvik executables, resource files etc. Dalvik executables are compiled byte codes. The application is stored in /data/app directory on Android OS and can be accessible only to root user for security.

1. JSch

JSch is a Java implementation of SSH2 based on JavaTM Cryptography Extension (JCE). This library allows to connect to sshd server, port forwarding, secure terminal emulation, secure file transfer etc. SSH2 supports secure file transfer, secure remote login, and secure TCP/IP and X11 forwarding. It automatically authenticates, encrypts, and compress transmitted data.

1. GSon

Google GSon library serializes and de-serializes Java objects to JSON and back. The library provides simple methods to convert JSON to Java object and vice-versa. The library is highly customizable and can take complex java objects. GSon can also work on arbitrary Java objects with no source code available

1. Firebase Cloud Messaging

The Firebase Messaging service by Google provides APIs for accessing Firebase Realtime database. APIs are available to support subscribing to topics and sending upstream messages, receiving messages from Firebase Cloud Messaging, and fetching Remote Firebase notification details. Firebase cloud messaging service has been used in the Mobile App to receive notifications about new client connections to the openWrt device.

### AmazeRT Agent

AmazeRT Agent is part of the AmazeRT software that runs on the OpenWRT router device. This is installed during the initial setup and will continue to run in the background, handling communication with the Cloud Backend.

1. Python with Websockets

Since OpenWRT Stack can run on a variety of hardware architectures, keeping the AmazeRT Agent independent of the underlying CPU architecture was required. To handle this, we chose to implement it using Python programming language. The AmazeRT Agent software's primary purpose is to handle the communication with the Cloud backend and process the requests forwarded to it from the Mobile App Client. This required a persistent communication channel to talk to the Cloud backend. For communicating with the Cloud Backend, websockets library was chosen since it provided a good infrastructure to handle custom communication protocols on top of the secure TLS layer.

Python also has extensive set of libraries that helps with rapid prototyping of the software, letting us focus on the actual functionality, rather than spending effort on the Lower-level libraries and utility functions. Though OpenWRT SDK provides more low-level libraries and frameworks for developing native applications for OpenWRT based devices, handling different CPU architectures required a lot more effort from developer's side. For the purpose of prototyping, we did not need such low level access to the OpenWRT stack. Due to these reasons, we chose not to use the OpenWRT SDK for the prototyping.

1. UUID

Every device that is managed by the AmazeRT system will need to be uniquely identified. A UUID was chosen as an identifier for the device. UUID is a 128-bit number, that can be generated to be uniquely without a central database of all generated Ids [6]. AmazerRT agent also uses UUID to generate a secret password that is shared between the device and mobile app, for securing and validating sensitive data sent across them.

## Middle-Tier Technologies

[…]

Cloud

App Engine

Cloud functions

WebSocket

Figure Cloud Architecture

### App Engine

App Engine is an platform as service infrastructure provided by Google to host an application with out worrying about instance management, scaling, resource allocation etc. It is a Linux container hosted in google public cloud. In this project App Engine instance is used as middle tire facilitating the communication between the Amaze RT device and the mobile application. App Engine takes care of authentication and authorization between Amaze RT device and Firebase Database.

For the communication between WRT device and mobile App an AppEngine (a cloud linux container instance) instance with public websockets is used. WRT devices can connect to this well-known websocket urls to asynchronously talk to the app. This architecture gives the flexibility that, even if the mobile app is not running at that time the WRT device can establish a connection to the websocket and push its message.

### Cloud Function

Cloud functions are serverless computing infrastructures which can be used to invoke a function for a specific event or a trigger. It is used to publish database modifications to App Engine websockets which in turn push the message to corresponding WRT device.

### Websockets with TLS for security for device to App Engine communication

## Data-Tier Technologies

### Secure Data Storage

Mobile App will utilize local device data storage for device specific information.

### FireBase RealTime Database

The Firebase Realtime Database is a cloud-hosted database where data is stored in JSON format. Every connected client to a Firebase Realtime database gets synchronized and gets data updates in real time. Firebase Realtime Database used for OpenWRT router to Mobile App communication including router settings and status notification. Firebase Realtime Database is used to implement the cloud-based device state and event management across managed OpenWRT devices and their managing applications. The Firebase infrastructure provides User authentication and authorization for appropriate access control. It also provides event trigger and registration mechanisms to help implement the business logic.

### File Encryption for App data storage

EncryptedFile class that is part of the androidx.security.crypto package can create and read encrypted files. The encryption scheme supported by the class is AES256\_GCM\_HKDF\_4KB. MasterKey class references a key that is stored in Android Keystore. The recommended master key size is 256 bytes. The key encryption scheme used by MasterKey class is AES256\_GCM\_SPEC. EncryptedFile class uses the MasterKey class derived Key to encrypt and decrypt the files.

### Secure Device to Mobile App data transfer

Device Settings data transmitted between the OpenWRT device and the Mobile App are at risk and needs to be secured. To accomplish this, Symmetric Key encryption shall be used to secure this device settings data, while in transit and as stored in the cloud.

Symmetric Key encryption will make use of the unique device registration Id generated during the initial device setup phase in deriving the Symmetric key shared by this device and its managing Mobile App. The encrypted device settings data is saved in the Firebase Realtime database after concatenating the IV, message digest and the cipherText to form a single byte stream.

# Project Design

## Mobile App

### App Login UI

A screen shot of a computer

Description automatically generatedThe launching page of Mobile App as presented in Figure 8 will direct to federated login using Firebase Authentication UI.

Figure App Launch page

Graphical user interface, text, application

Description automatically generatedThere is no sign-up feature as the app is Android based and all users will have Google account for login. The federated login screen UI is presented is Figure 9. The user needs to provide the Google username and password only the first time. The user gets navigated to Devices screen only on successful login.

Figure App Federated Login

### Device Registration UI

Successful login will launch the Device list screen. This screen will list all the registered OpenWRT devices as presented in Figure 10.

Figure Device List screen with fab button to add device

Graphical user interface, application

Description automatically generated

First time use of the app will not have any device listing. The fab button (with plus sign) on the Device list screen will navigate to Add device screen as presented in Figure11.

Graphical user interface, text, application, chat or text message

Description automatically generated

Figure App, Add device for registration

The user needs to provide the device name which the OpenWRT device needs to be remembered. App will do an input check if a device with same name is already registered. It will proceed only with a new valid device name. User needs to provide admin username and password of the device to be registered.

Graphical user interface, application

Description automatically generatedAdd button will start the registration process. A circular progress bar will be displayed along with the text running display of the registration sequence as presented in Figure 12. After the device registration is done, the Done button will appear on the same screen as presented in Figure 13.

Figure App, Device Registration Progress

Graphical user interface, application

Description automatically generatedDone button will take the user back to Device list screen. This screen lists the currently added device along with all OpenWRT devices registered before as presented in Figure 14.

Figure App, Device registration Done

Graphical user interface, application

Description automatically generated

Figure App, Device List with newly added device

Figure 15 depicts the device registration sequence from Mobile app. The device registration sequence is as follows.

1. ssh to OpenWRT device using JSch library and execute commands to install secure ftp daemon and start the daemon.

2. Securely copy the device package that is bundled in Mobile app to device using SFTP.

3. ssh to OpenWRT device, untar the device package and execute install script on the device.

4. Securely copy the device info json file from OpenWRT device to Mobile.

5. Encrypt the device registration json with Android MasterKey and save in App data space.

6. Register the OpenWRT device on Cloud using Firebase database.

7. ssh to OpenWRT device and start the device software.



Figure Device Registration Sequence Diagram

### Device Settings UI

Mobile App provides support for managed OpenWRT device settings configuration. This acts as a user’s primary access point to control the features provided and exposed by the OpenWRT device. The settings screen follows the device registration control flow by an authenticated user for App registration. At this point in the control flow, the authenticated user’s Unique Identifier and the registered device’s unique identifier are generated and available. These Unique Identifiers that are encrypted using the device symmetric key are decrypted and stored in process memory. The settings UI is designed using the MVC design paradigm. A combination of recyclerView, viewModel and adaptor is used to capture and display the device settings data using a livedata model with observer.

1. Settings fetch

The settings data is fetched from the Firebase Database that holds the current device settings data and acts as a communication proxy between the OpenWRT device and the Mobile App. A Firebase Livedata will be fetched using the built-in listener class. The listener class links to the firebase url and settings path referred to by the listener object. This allows for instant updates to be delivered from the firebase Realtime database to the registered mobile application. The fetched data from the Firebase Realtime database will always be the latest value for each setting updated from the corresponding device.

1. Settings push

The settings data reconfiguration is facilitated through a child UI screen. Each setting update happens over a child screen that supports single setting reconfiguration. User is presented with all possible options for the current setting that is selected for update. Once user confirms update of the setting, the same is updated using the setting path reference and of the Firebase Realtime database url.

### Device Status UI

Mobile App provides support for user to view current device status. The status UI screen is designed using the MVC design paradigm. A combination of recyclerView, viewModel and adaptor is used to capture and display the device status data using a livedata model implemented with an observer. Firebase Realtime database proxy provides the live status data updates over a dedicated status path for each device. Status UI captures the details of devices that are connected to the OpenWRT device including the burned in MAC address. OpenWRT device status that captures the current running status of the device is also captured here along with device bootup time.

### Securing data in transit

All the settings data transmitted between the OpenWRT device and the Mobile App are encrypted using Symmetric Key encryption. This ensures that critical settings information of the OpenWRT device is secure in transit as well as while at rest in the Firebase Realtime database.

The unique device registration Id generated during the initial device setup phase is used in generating the PBKDF2 for deriving the Symmetric key shared by this device and its managing Mobile App. The Symmetric key uses AES GCM mode and is derived using the secure key parameters: Key length (256), Iteration count (100000). A random generated initialization vector is used to further strengthen the encryption.

Once the settings value has been encrypted using the Symmetric key, the IV, message digest and the cipherText are concatenated to form a single string as in Fig: below and stored in the Firebase Realtime database.

Fig: Packing IV|MessageDigest|CipherText

### User Profile UI

Graphical user interface, application

Description automatically generatedMobile App has a Toolbar on all screens once logged in. The Toolbar has user profile icon on the right corne. Google profile photo is displayed on the user profile icon. User click on user profile icon will take to User profile screen as presented in Figure 16.

Figure App User Profile screen

This screen provides option to logout from the current user login. Logout will take the App back to the launch screen to login.

## AmazeRT Agent

AmazeRT agent is the part of the AmazeRT software stack that runs on the OpenWRT router. It starts as soon as the router boots, and keeps running until shutdown. The primary purpose of device software is to handle all the communication between the device and the Cloud backend service. It periodically sends the device’s settings and current status to the Cloud backend and also handle different requests originating from the Mobile App, forwarded by the Cloud backend. The agent is designed to run on any device that support OpenWRT software stack.

### Registration and Installation

The AmazeRT agent software is designed to be installed on the OpenWRT router by any user who has administrative access to the device. Since the software is designed to be independent of the the OpenWRT SDK and associated ecosystem, we package this along with the mobile app used as a client for the entire software stack. The basic sequence of device setup includes copying the installation package to the router, running the install scripts to install and generate device id and keys, and then restart the router.

There are two installation scripts provided in the package. The first one, *“install.sh”* is a shell script which is used for ensuring all the required dependencies are met. It uses the OpenWRT package management system called opkg to install all the required packages, including python, websockets, cryptography libraries. Once all the dependencies are installed, the AmazeRT agent files are copied to the right locations. It also setup the router to start the AmazeRT Agent on boot, making sure the communication with Cloud backend is functional across reboots.

### Cloud Registration

When the AmazeRT agent starts, if it is able to communicate with the Cloud backend, we assume that the router is in a consistent state. Any data in the cloud about the settings may be different from what is on the router. There is no guarantee that these settings are in a consistent state. To prevent such conflicts from corrupting the device state later, we will need to overwrite all that settings in cloud to the one from the router. Once the agent starts, it prepares an initial registration packet which contains the all the supported settings and current status. This packet is then filled with the device identification data, and all the settings are encrypted and signed for security, and then sent to the Cloud backend. Once this completes, the state in router and the Cloud backend are in sync. The mobile app will be able to pull the data from the cloud database. The registration packet is only sent once during the boot, when the agent starts. If the agent is restarted for any reason, a registration packet is resent with the updated data. This is to handle the case where one or more setting might have changed thorough other management interfaces like LUCI or command lines.

### Heartbeats and Status Updates

The registration packet sent initially will have a complete set of settings and status from the router. The settings may change due to updates triggered from other interfaces like LUCI or even command line settings. The router status also may change due to Wifi clients connecting or disconnecting from the device. The AmazeRT agent will keep sending such updates to the cloud backend at a fixed interval so that the mobile app UI can be updated with the right data. These packets are referred to as “Heartbeat packets”. The structure of the heartbeat packet is similar to the registration packet, except that the settings are filtered to remove those that did not change from last time it was sent to the cloud backend. Note that the setting values are encrypted and signed, each time with a different nonce also added, so even if the setting value did not change, sending the setting to cloud will trigger a change in the database. This filtering reduces the number of setting updates that need to be handled from the mobile app side too.

### Device Configuration Management

Mobile app may change the value for any setting that is supported by AmazeRT, by updating it in the cloud database. The cloud backend will process the change in value for the the setting and generate a setting request packet to be sent to the router. The packet shall have the device identification data and sent to the router. AmazeRT agent will then process this request and extract all the settings embedded in the packet and apply those settings locally. Once the settings are updated, the heartbeat packet will take care of sending the update back to the cloud. If for any reason the setting is rejected, the next heartbeat is forced to sent a full update, so that the cloud database is restored to a valid state. This change will also cause the Mobile app to update the device settings on the UI.

In addition to handling setting changes, AmazeRT agent supports a “command” packet to execute any generic command on the router. This helps implementing support for device reboot, uninstallation of AmazeRT agent, or any other special updates using this framework.

### Failure Handling

The AmazeRT software stack relies on the AmazeRT agent running on the router, handling all the communication with cloud backend. The AmazeRT agent also requires a persistent connection to the cloud to handle communication. However this cannot be guaranteed due to the possibility of network disruptions etc. To handle these kind of issues, AmazerRT agent is run with a lightweight wrapper called “runner” which keeps restarting the agent if the main application stops for some reason. The main application is then written to gracefully exit if there is any communication failure with the cloud, or any unexpected error during its execution. The runner then restarts the agent and re-establish connection with the cloud backend.

## Cloud Backend

### Websocket

### AppEngine

### Cloud function

## Secure Data Storage

### Shared Configuration Database

Firebase Realtime Database used for OpenWRT router to Mobile App communication including router settings and status notification. Firebase Realtime database is used as a proxy for real time data update between the OpenWRT device and the Mobile App. The real time notification and registration capabilities provided by this Google service helps the Mobile App and the OpenWRT device to communicate through an effective and secure channel.

Data Access Control

Firebase Realtime database rules have been used to achieve data access control for the device data at rest. Write and read access to the data stored and synced in the database is restricted to the authenticated user using the auth rule option. Firebase Authentication feature has been used as the authentication method for user verification and later extended in data access control. User authorizing based on their Firebase authenticated identity is used to ensure and restrict read/write access of device data.

### Device Registration data on Mobile app

The registration info received as json file from the OpenWrt device is encrypted and saved in the App data space of the mobile device. The json file is encrypted for security as it contains device specific information. It is saved locally on device for persistence of data. The file name used to save the data is the device name user inputs for the OpenWrt device on registration, concatenated with “. dev”. When user navigates to the Device screen from device list screen, the file name is constructed by concatenating the device name with “. dev”. This file name is used to fetch the device specific json file and retrieve the contents using Gson library.

# Project Implementation

## Mobile App

### App Login UI

This component is responsible for federated Auth login. Firebase AuthUI and IdpResponse classes that are part of com.firebase.ui.auth package are used to implement the federated login. On AuthUI Sign in request the App passes requestCode of ‘1001’. On IdpResponse the App checks the requestCode and if that matches ‘1001’, will check if Signing in was successful and proceed navigating to the Device List activity.

### Device Registration UI

This component is responsible for adding a new device to the App. The following classes are used to implement the different functionalities required for Device Registration.

* SshInitTask class
* SftpTask class
* SshTask class
* SftpGetTask class
* SshPostTask class

All the above-mentioned classes are doing network activity that are blocking in nature. App cannot block the main thread for blocking activities. So, all these classes are derived from AsyncTask class that spawns a new thread. Each of these classes overrides two functions of AsyncTask class, doInBackground() and onPostExecute().

doInBackground() overridden function for all the above-mentioned classes have JSch() class object that starts a secure session with the OpenWrt device with the provided admin credentials. Classes SshInitTask, SshTask and SshPostTask opens a ChannelExec channel to execute the remote commands. Classes SftpTask and SftpGetTask opens a ChannelSftp channel to copy files to and from the remote OpenWrt device.

After collecting OpenWrt admin credentials from the Device Registration UI, App will execute the SshInitTask. This class is responsible for executing remote command on OpenWrt device to install sftp daemon. onPostExecute() of SshInitTask calls SftpTask. This class is responsible for securely copying the device package from Mobile App to openWrt device. onPostExecute() of SftpTask executes SshTask. SshTask is responsible for secure execution of commands on the OpenWrt device. The commands are un-taring the ‘amazert’ package, start the install script. This will in-turn install the required packages in OpenWrt device, do OpenWrt device initialization, and creates the device registration json file. onPostExecute() of SshTask will start SftpGetTask. This class will securely copy the device registration json file from remote OpenWrt device to Mobile App.

The device registration json file gets Encrypted using EncryptedFile class from androidx.security.crypto package. MasterKey class will pick Android device key to do the symmetric encryption of the json file and gets stored on the App data storage location that is not accessible to other apps or users.

onPostExecute() of SftpGetTask registers the device on cloud. Firebase.database class from com.google.firebase.database is used to set the Cloud database with the details from current logged in user and device registration details. After Cloud registration SshPostTask is called. This class will execute the device registration to cloud script on the OpenWrt device and start the amaezrt daemon on the OpenWrt device for all further communication of the device to Cloud.

From the time the first async task is called till the last registration step, the front-end registration UI will show the circular progress bar along with a text view of main milestones on the registration.

### SymKeyEncryption class

This class is used to provide the symmetric key encryption infrastructure. It initializes the symmetric key shared between the OpenWRT device and the Mobile App using the unique registrationId generated as part of the initial device setup handshake. It exports APIs to:

1. Generate the Initialization Vector

2. Encrypt input plaintext using Symmetric Key Encryption

3. Decrypt input ciphertext using the shared Symmetric Key

4. Decode the padded cipher text to segregate the IV, Message Digest and Cipher text.

The key encryption scheme used by SymKeyEncryption class is AES256\_GCM and uses secure key parameters Key length (256), Iteration count (100000) and a random generated initialization vector to further strengthen the encryption.

### FirebaseQueryLiveData class

This class is used to add event listener for Firebase Realtime database data change events. This class is derived from the LiveData Data holder class to allow for observing the changes in Database data snapshot. This class is designed and implemented in a generic fashion to instantiate and query device settings or device status database paths.

The instantiated object of this class feeds into the DeviceSetting and Device Status ViewModel class instances.

### Device Settings UI

This component is responsible for device settings management, allowing the user to view and update OpenWRT device settings. The Mobile App user authenticated using Firebase authentication APIs can choose to navigate to this UI component. Settings specific to this user and the selected device are displayed with options to edit the same. FirebaseQueryLiveData object is initialized with the Firebase Realtime database settings URL for the mapped user’s device.

### Device Status UI

This UI component is responsible for fetching and interpreting the managed OpenWRT device status. The Mobile App user authenticated using Firebase authentication APIs can choose to navigate to this UI component. Settings specific to this user and the selected device are displayed with options to edit the same. FirebaseQueryLiveData object is initialized with the Firebase Realtime database status URL for the mapped user’s device.

### User Profile UI

This UI is responsible for showing the logged in user profile on Toolbar and navigate to User profile screen on clicking the profile icon. User profile screen provides Logout option. ProfileDownload class derived from AsyncTask class is responsible for downloading and saving the profile picture of logged in Google user. BitmapFactory class is used internally in ProfileDownload class to convert the profile picture to bitmap to show as icon on Toolbar.

## AmazeRT Agent

AmazeRT agent is implemented using a combination of python and shell scripts. It is implemented as an app which runs forever, sending information from the router to the cloud backend, and processing requests sent from the Android App via the cloud backend. Websockets library is used to implement the communication channel between the agent and the Cloud backend. All communication between the agent and cloud are JSON Objects which describe the action to perform, and the parameters for the specified action. The following sections describe the details of these actions and their intended purposes.

### Registration and Installation

The AmazeRT Agent is installed on the OpenWRT router while setting up the device from the AmazeRT Mobile app. The Mobile app is packaged with the AmazeRT agent installer. During the initial setup of AmazeRT, the installer is copied over to the router and the install script is executed. The install script will ensure that the executables required for AmazeRT Agent is copied over to the right locations. A startup script named “amazert” is configured to be run at the end of the boot cycle. This script will start the the “runner” script which is responsible to ensure that the AmazeRT agent is running all the time in the background.

The AmazeRT system requires every device to be identified uniquely, and associated with a specific userId. The “init” script in the software package is used to generate this information and save this on the router for later use by the agent. This information is saved in the file “/etc/amazert.json”. The structure of the file is as described below

|  |
| --- |
| {  “registrationId”: <uniqueRegistrationId>,  “email” : <emailIdOfTheRegisteredUser>,  “uid”:<FirebaseAuthenticatedUID>,  “deviceId” : <UUID of the device>  } |

RegistrationId is a uuid generated for the purpose of symmetric encryption of data between the AmazeRT agent and the mobile app. This is generated and exchanged during the initial setup. A key is generated from this by the AmazzeRT Agent and the mobile app later for encrypting and signing the communication over the cloud backend.

Email is used only for identification and for debug purposes. The uid is used for actual authentication purposes when the Mobile app communicates with the database and the cloud backend. These values are sent from the Mobile app, once it authenticates with the Cloud, when the init script is run.

deviceId field contains a unique ID generated by the init script during the setup. This is used to identify the device uniquely in all the communication between the Mobile App, Cloud backend and the AmazeRT Agent.

### Cloud Registration

The registration packet from AmazeRT agent uses the json object of the following format.

|  |
| --- |
| {  {  "action": "register",  "settings": [{  "name": <name of the setting>,  "value": E(settingValue, K)  },  .... Other settings follows...  ],  "status": [{  "name": <name of the status>,  "value": <value of the status as json  },  .. Other status follows...  ],  "identifier": {  "uid": <uid from registration>,  "deviceId": <deviceId>,  "email": <email address as string>  }  } |

The settings in the packet is always encrypted using the AES encryption with 256 bit key in GCM mode. A cryptographic nonce is also added for increased security. The key K is derived from the registration Id when the AmazeRT agent starts. The encrypted setting is also added with the message digest so that when it is decrypted at the Mobile App, it can be verified. Status values are not encrypted for now since they are not sensitive as the settings. For example the Wifi Password setting is a sensitive data, which if exposed unencrypted can be a security issue. IF there is a data breach in the cloud database, since the settings are now encrypted, the Wifi password is still secure. Since the registration Id is not shared with the cloud backend or the database, the breached data is still secure until the attacker can decrypt it using brute force attacks or the attacker gains access to the registration Id of each device registered with the cloud. This lead time helps to re-register the devices (once we detect the data leak) and change the secure settings like Wifi password as needed. Not all settings need to be secured, but for the prototyping phase, keeping all values encrypted proved to be simple to get this ready within the short amount of time we had.

Settings and status follow a generic structure with name and value as two required fields. The value may be a JSON or a string depending on the requirement. The names for settings are kept as the same as those used in the UCI interface in OpenWRT. For status, we created separate unique names specific to AmazeRT.

Settings and Status values are fetched using a generic framework to describe how a value can be fetched from the system. The Settings framework uses an array of JSON objects, each for a specific setting , and contains the command that can be used to get or set the value, a prologue and epilogue for the get or set command, a filter that can be used to restrict possible values, and a default value that can be used if the get failed. Status framework uses a simplified version of the settings framework since they are read only. Each status is mapped to a command that can provide the JSON object that need to be returned as the status. The command may be a built-in command, or a custom shell script or python script implemented as part of AmazeRT agent. This method allows to expand the number of status and settings supported by the software with minimal effort, and still work with older versions of Mobile apps, without version incompatibilities.

### Status Updates and Heartbeats

Once the registration is complete, AmazeRT agent starts a separate thread which keeps sending the heartbeats and status updates at fixed intervals. However since the settings are not going to change between every heartbeat, a filtering mechanism is implemented in the agent to remove the unchanged settings before sending them to the Cloud backend. The comparison is done pre-encryption values, so as to prevent noise introduced by the nonce triggering frequent setting updates. Status values however are sent unfiltered, since they are more prone to change. Implementing a filtering support is planned as a future update, to minimize the amount of data exchanged between agent, cloud and mobile app.

|  |
| --- |
| {  "identifier" : {  'email': 'nabin@gmail.com',  'uid': '\_SDFsEfRSDjFCZXCVASEf',  'deviceId': 'fb967061-168a-11eb-9272-88e9fe6b97d6'  },  "action": "setting",  "setting" : {  "name" : "wireless.wifinet0.ssid",  "value": "tT760DVgoZtDjjkpHyIALw==bHuwfcQVlaO6C5wt6Udw4w==Q8pD3w=="  }  } |

### Device Configuration Management

The settings supported by AmazeRT agent are modifiable by the Mobile App. The modification is always done in the mirrored data in the cloud database. The cloud backend listening to these changes will forward that change to the right AmazeRT Agent instance based on the device uuid. AmazeRT agent, once it finishes the registration will run an infinite settings handler loop and look for messages received from the cloud backend. Each message is a JSON object, and is expected to have an "action" field which describe what action need to be taken upon receiving the message. The action type "setting" is used for applying a new setting to the device. For every action packet of "setting" type, a field "setting", provides an array of Json name/value objects as in case of the registration packet, describing the settings that need to be applied on the router. AmazeRT agent implements a setting handler function which will look up each of the setting name against the internal rules, validate it and run the prologue, command and epilogues as needed to apply the setting. If any setting is rejected, we invalidate the last settings sent to the cloud backend, so that the next heartbeat will send the entire settings and make sure that all the settings in the database and hence the Mobile app are in a consistent state. A sample packet from the cloud backend to change a setting on the router is listed below.

In addition to the setting action, a "command" action is also supported, which can run a shell command on the router. This may be used by the Mobile app to execute command like reboot on the router. The command is still routed via the cloud, which also provides another layer of security with authentication. At the moment the command is assumed to be from a trusted entity, since it is encrypted and signed using the symmetric key generated from registration id, in the same way as setting values. However, in the future, it will be restricted to an explicitly allowed list of shell script that are built as part of AmazeRT agent. The framework allows such a behavior change with minimal impact and effort to make it secure.

### Failure Handling

The AmazeRT software stack relies on the AmazeRT agent running on the router, handling all the communication with cloud backend. Any failure in the agent is going to be a major issue, since the device no longer can be controlled or recovered without physical access or direct SSH access to the router, bypassing the agent. This will defeat the whole purpose of having the AmazeRT system for managing the router. To provide such reliability, we have multiple levels of failure handing in the AmazeRT agent.

At the Lowest level, each commands and status processing is wrapped with exception handlers to handle any issues with invalid commands or broken message packets. When an invalid command is received or a setting update fails, we still need to make sure that the Mobile app is aware of the failure. However since the mobile app does not directly communicate with the AmazeRT agent, it is not possible to send a reply to the command and hope it would reach the app. To handle this, the agent invalidates its local cache of last settings sent to the cloud. This forces the heartbeat to send all the settings next time to the cloud without filtering anything. Any setting that failed to apply on the device will so be reverted to a good value that reflect the current state on the router. The Mobile app will get notifications from the cloud database about these changes and it can handle updating the UI to show the reverted settings. The AmazeRT agent also handles communication errors and errors during get/set of settings and status values in a similar way.

Settings that are sent from the cloud may be attempting to set some settings to an invalid or unsupported value. To avoid this case, we have a provision to add rules for possible values for a setting. Any value that falls outside the allowed list may be rejected by the system, and a recovery process will take place to refresh the cloud data using another registration packet.

Other type of errors may still cause the agent to be terminated. For example, if the Cloud is unreachable for some time, the connection will need to be reset. Doing this immediately might not recover, since network issues may take time to recover. To handle these cases, we terminate the AmazeRT agent gracefully, updating the logs. The wrapper script called "runner" will take care of restarting the AmazeRT agent after a small interval to prevent thrashing the router's CPU too much. Since the agent is restarted, it goes through the full registration process with the cloud and recover the state of the router reflected in the cloud and the mobile app.

The runner script is designed to be as simple as possible to minimize the chance of it failing and crashing. This provides more reliability to the entire software stack running, as any failure in the AmazeRT agent, or communication with the cloud is recovered automatically in a few seconds of time.

## Cloud Backend

### Websocket

### AppEngine

### Cloud functions

## Secure Data Storage

### Shared Configuration Database

### Device Registration data on Mobile app

As part of Device registration the App receives amaezrt.json json file that contains the unique device id , registration id along with user info. EncryptedFile class that is part of the androidx.security.crypto package is used to encrypt the json file and store in local App data storage. The encryption scheme used is AES256\_GCM\_HKDF\_4KB. The symmetric key used to encrypt is given by MasterKey class which takes the key Android Keystore. App uses the recommended master key size of 256 bytes. The key encryption scheme AES256\_GCM\_SPEC is used. The same key is used to decrypt the file for App use

## Hardware platform

Raspberry Pi 4 was chosen as the HW platform for implementing the router side of software for ease of debugging and availability of enough RAM for running debug tools as needed. For the Android App side, a simulator running Pixel 2 was chosen as the base hardware for prototype testing.

# Testing and Verification

Describe your test strategy, process, and results for verifying the functionality of your project.

## Testing Process

### Mobile App

1. Database rules validation

Firebase Realtime database rules are used for data access control. The derived rules were tested using a test UI screen with the capability to fetch and derive database URLs based on the passed in values to verify the access controls in place. Database rules were specifically tested for the read and write controls added. Automated tests for this Firebase Realtime database module are being worked upon now.

1. UI

All the UI tests are done manually. Device Registration UI has inputs that need input check validations. Different range values are used to test the input fields of Device Registration UI. Negative test cases of Device Registration are done by switching off the OpenWrt device and trying the registration. The App should gracefully handle network issues and report it correctly on UI. All other UI screens are visually inspected for correct behavior as expected.

### AmazeRT Agent

1. Installation

Installation is completed by the install script. Module level testing for this was done with manually following the package copy and install script steps. Once the installation is complete and restarted, the runner script should be started, which can be verified by looking at the logs in /var/logs/amazert.log.

1. Initial configuration

The init.py script does the initial configuration. It accepts the UID and email ID as the params and creates the configuration in /etc/amazert.json, if it does not exist. As negative test case, retrying the same should not change the content in the configuration. However if the configuration is deleted, then the next invoke should recreate a new configuration, even if the same email id and uid are supplied. The script is also tested with missing parameters, when it should fail the creation of configuration.

1. Cloud Registration

For module level testing, a test tool which can fake the cloud backend using a local server and an HTML based client was implemented. Upon starting the agent, The registration packet is sent to the cloud, which will be instead sent to the local server for testing, and the HTML client will show the raw data sent across.

Once the module level testing was completed, another round of testing with the corresponding changes on Cloud backend and Mobile app were used for end to end testing to validate the device registration.

1. Device Configuration Changes

The HTML based test tool mentioned earlier was used to send configuration changes to the device using raw JSON data. The results were observed when they are sent back to the local server and compared against expected value.

Setting changes were also tried with invalid values. All settings updates testing for module level testing was done with encryption disabled to make it easier to create hand created JSONs for testing. Once the module level testing was completed, another round of testing with the corresponding changes on Cloud backend and Mobile app were used for end-to-end testing to validate the setting updates.

1. Failure Recovery

The AmazeRT agent is supposed to recover automatically from failures. Following scenarios were tested to ensure this.

1. Sending invalid configuration values to the device from the client.( Mobile app/test tool)
2. Sending corrupt data to the client by corrupting the cloud database
3. Sending corrupt commands to the client by corrupting the cloud database.
4. Sending commands encrypted for a different device, by manually editing the cloud database.
5. Forcing a network disruption to check if the agent recovers and reregisters itself with the cloud
6. Forcing the Agent to be killed using OpenWRT commands to terminate the process and confirm that the agent restarts
7. Ensuring that after an uninstall, the agent no longer starts on the router.

### Cloud Backend

### Database testing

## Test results

# Performance and Benchmarks

Describe any performance and benchmarking criteria you used for your project. In addition, describe any benchmarking results you observed in your project.

## Scalability

## Throughput

## Reliability

## Feature Parity

# Deployment, Operations, Maintenance

## Cloud Backend

### Cloud Functions

Cloud function is deployed in google cloud. When the cloud function is upgraded with new features google cloud updates it reliably without down time.

### App Engine

App Engine is deployed in google cloud as a compute instance. It provides a public IP and a persistent websockets based connection to the managed devices.

## Android App and Device software

Deployed on mobile per device, used for managing the device.

### Android App Stores

### Packaging device software

## Secure Database operations

### Firebase Realtime Database

Firebase RealTime database is used for the AmazeRT device to Mobile App communication.

### Local App database

### Device data storage

## Github repository

Contains all project artifacts.

# Summary, Conclusions, and Recommendations

## Summary

When the project was initially conceived, it was intented to be a full fledged management tool for OpenWRT based routers. However upon research we found that original scope for this project was too big to be completed within the timeframe we had, but we were able to trim down the set of features supported and make the end-to-end flow working. A framework was implemented to add new settings support with minimal change from the AmazeRT Agent that runs on the OpenWRT router. The Mobile App UI would require more changes but can be redesigned to make it more data driven.

Our original design was to use a Custom Linux server running on AWS for the Cloud backend logic processing. However, considering the limited amount of work that is done by the cloud backend and the ease of integration with other modules, use a Google App Engine, Firebase and Cloud functions was chosen. This change also allowed us to remove the requirement of exposing a public IP Address for the OpenWRT router for the cloud backend to push commands to the router.

We have spent considerable amount of time to do the background research for implementing a system for secure remote management of routers. This helped us understand the potential points of failures in terms of security, and solutions for those issues. The way we have designed the project is such that this does not have to be a management tool just for routers, but for any Edge computing device. Although the project as is cannot be used for this purpose on a different edge computing device, the design can be applied, and a majority of the agent and cloud backend implementations can be reused for implementing such a solution

## Conclusions

1.Achieved all the revised project goals and implemented some of the stretch goals as well.

## Recommendations for Further Research

TODO: Reformat the content

Enhancement of this infrastructure to support device management for Edge computing devices, using Mobile app and the framework provided by the AmazeRT cloud and device management modules

* Extend with IOS application support with SwiftUI for User interface development
* Moving from Firebase to Firestore for the database
* Moving to Android Jetpack compose to design modern scalable UI.
* Redesigning the Android app for generating configuration and status screens using a data driven mechanism so that new features can be added with minimal efforts.
* Integrating this entire software suite with the open source OpenWRT codebase and making this a default install option for all OpenWRT devices.

# Glossary

|  |  |
| --- | --- |
| **Acronym** | **Description** |
| API | Application Programming Interface |
| JSON | JavaScript Object Notation. JSON is a lightweight data-interchange format. It is easy for humans to read and write as well as for machines to parse and generate. |
| XML | Extensible Markup Language. |

Table Glossary

# References

[1]     "Router (computing)", *En.wikipedia.org*, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Router\_(computing). [Accessed: 05- May- 2020]

[2]  Instructables.com, 2020. [Online]. Available: https://www.instructables.com/id/AndroidiOS-App-to-Access-Your-OpenWRT-Router-Remot/. [Accessed: 30- Apr- 2020]

[3]   "OpenWRT Project: Packages", *OpenWRT.org*, 2020. [Online]. Available: https://OpenWRT.org/packages/start. [Accessed: 01- May- 2020]

[4] "OpenWRT Packages", 2020. [Online]. Available: https://OpenWRT.org/packages/table/start. [Accessed: 05- May- 2020]

[5]"OpenWRT Project: Use SSH to connect to the internet and install Luci Web interface", *OpenWRT.org*, 2020. [Online]. Available: https://OpenWRT.org/docs/guide-quick-start/ssh\_connect\_to\_the\_internet\_and\_install\_luci. [Accessed: 02- May- 2020]

[6] “Universally unique identifier”. https://en.wikipedia.org/wiki/Universally\_unique\_identifier, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Universally\_unique\_identifier [Accessed: 02-Nov-2020]

# Appendices