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

APPROVED

Prof. Kaikai Liu, Project Advisor

Dr. Xiao Su, Director, MS Software Engineering

Dr. Rod Fatoohi, Department Chair

ABSTRACT

aMAZE-RT: Secure Management for OpenWrt
By Ginto George, Binu Jose, Sandeep Panakkal, Nabin Thomas

A typical home internet connection consists of a modem that connects to the Internet Service Provider and a Wireless Router that provides an Access Point to this network for home computers, mobile, and smart home devices [1]. Commercially available wireless routers have a locked firmware with restricted functionality, mostly managed using a web-based application running on the router. Some newer routers do provide mobile app-based management [13].

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 [12]. However, there is no serious open-source alternative to provide mobile app-based remote management of routers running stock OpenWrt firmware [2]. Having such a system would be a great addition to the OpenWrt ecosystem.

In this project, we have prototyped an end-to-end secure software to manage routers using a mobile application for Android. The system consists of a mobile app, as the user interface for remote management, a custom software installed on the router during the initial setup of the router, and a cloud-based communication system, enabling seamless management and monitoring of the device. A stock build of OpenWrt is installed on a Raspberry Pi 4 device to convert it into a home router. The entire system is designed to be portable across any OpenWrt-based router, as long as the user has administrative access to the device to do the initial connection and setup. Once the initial setup is complete, sensitive data is encrypted using a secret key that is shared between the mobile application and the router, so that such sensitive data cannot be viewed by someone having access to the backend software running in the cloud.

Acknowledgments

The authors are deeply indebted to Professor Michael Tjebben and Advisor Professor Kaikai Liu for their invaluable comments and assistance in the preparation for this project.

Table of Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1. Project Overview

Introduction

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 NetworkFigure 1 shows how a typical home network connects to the internet using a wireless router.

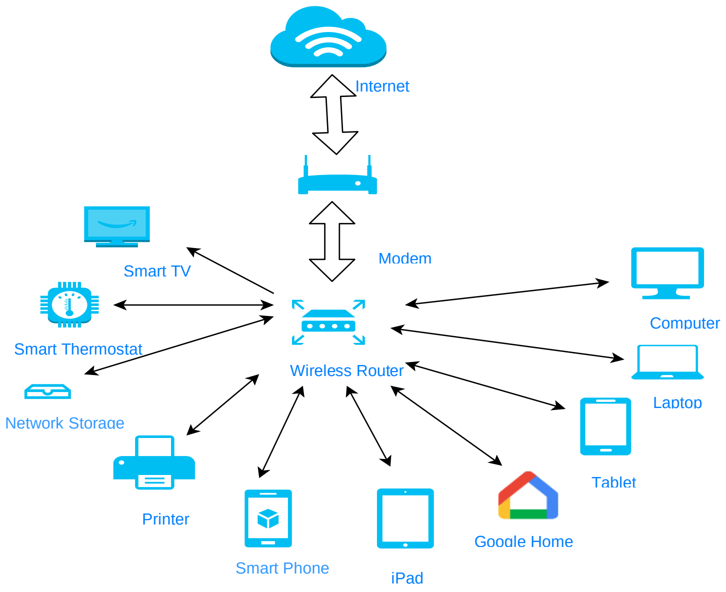


Figure 1 Typical Configuration of Home Network

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].

Figure 2 gives a high-level view of the components and their interaction in the AmazeRT management system.

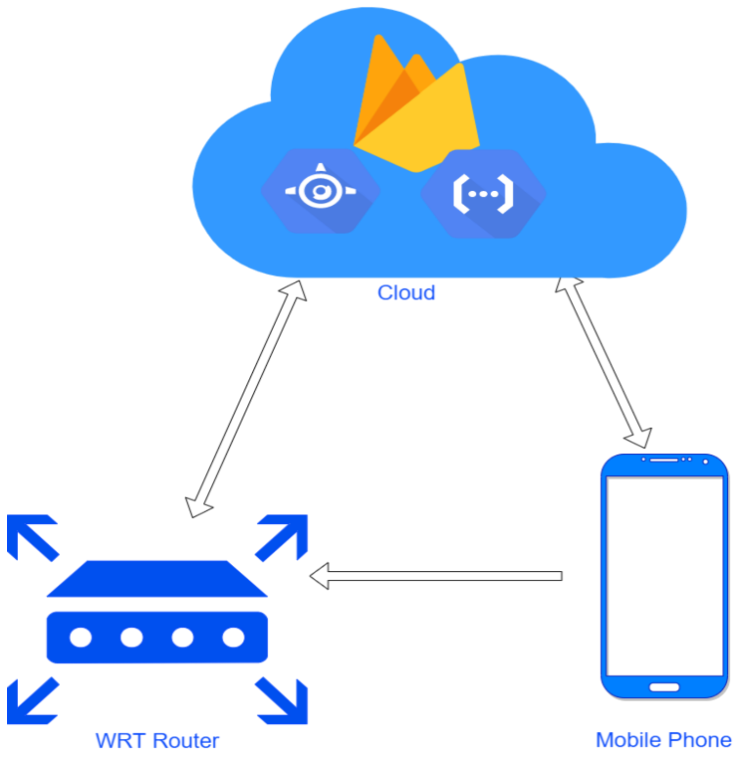


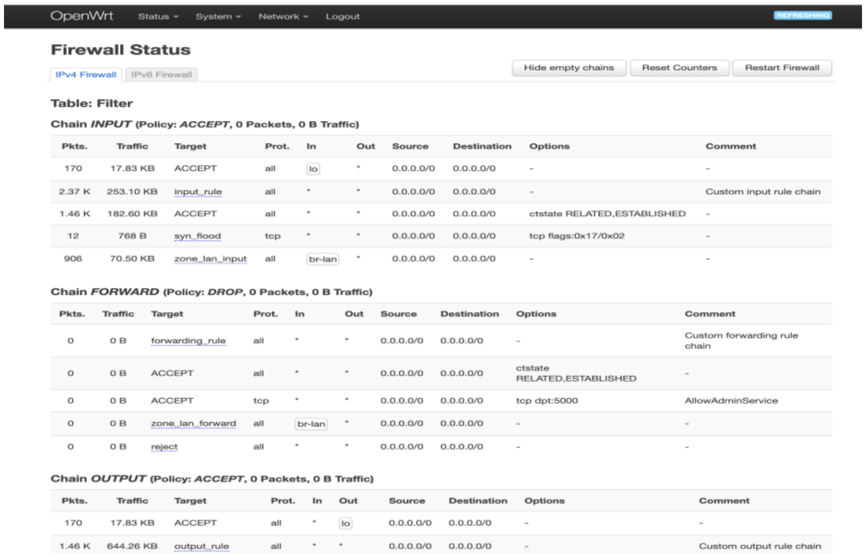
Figure 2 High Level Components

Proposed Areas of Study and Academic Contribution

OpenWrt is a very powerful software stack for routers. However, this lacks the modern App based management and monitoring system, which is getting increasingly common with commercially available home wireless routers. Being an open platform, OpenWrt allows 3rd party developers to add new features on top of the base software stack. Having a mobile app-based configuration management and monitoring tool will make OpenWrt based devices to enter the mainstream rather than being a playground for enthusiasts. In this project we propose to investigate the possibility of a cloud-based management and monitoring system, with a mobile app front end for managing OpenWrt based router with a simple and user-friendly way.

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 User Interface (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



Firewall Status

Table: Filter

Chain *INPUT* (Policy: *ACCEPT*, 0 Packets, 0 B Traffic)

Pkts.	Traffic	Target	Prot.	In	Out	Source	Destination	Options	Comment
170	17.83 KB	ACCEPT	all	lo	*	0.0.0.0/0	0.0.0.0/0	-	-
2.37 K	253.10 KB	input_rule	all	*	*	0.0.0.0/0	0.0.0.0/0	-	Custom input rule chain
1.46 K	182.60 KB	ACCEPT	all	*	*	0.0.0.0/0	0.0.0.0/0	ctstate RELATED,ESTABLISHED	-
12	768 B	syn_flood	tcp	*	*	0.0.0.0/0	0.0.0.0/0	tcp flags:0x17/0x02	-
906	70.50 KB	zone_lan_input	all	br-lan	*	0.0.0.0/0	0.0.0.0/0	-	-

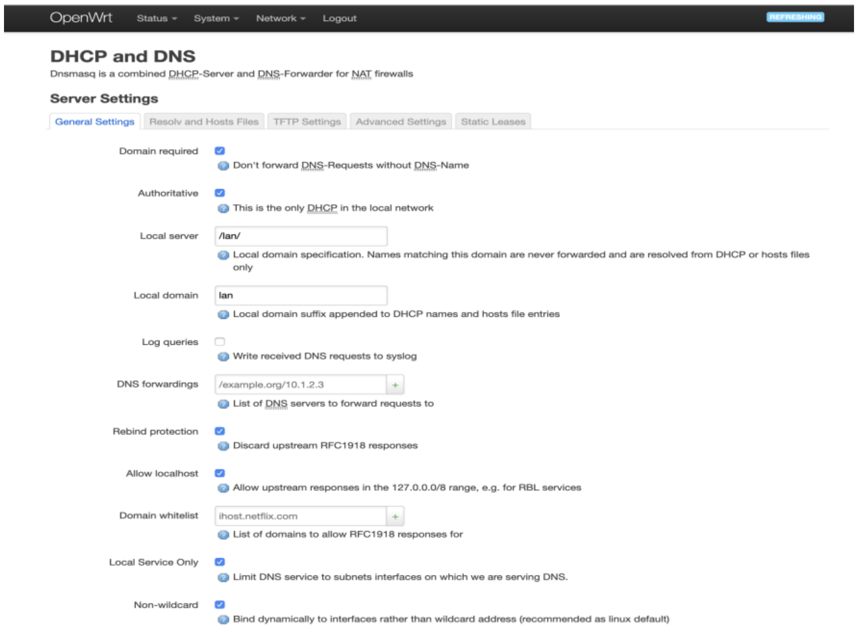
Chain *FORWARD* (Policy: *DROP*, 0 Packets, 0 B Traffic)

Pkts.	Traffic	Target	Prot.	In	Out	Source	Destination	Options	Comment
0	0 B	forwarding_rule	all	*	*	0.0.0.0/0	0.0.0.0/0	-	Custom forwarding rule chain
0	0 B	ACCEPT	all	*	*	0.0.0.0/0	0.0.0.0/0	ctstate RELATED,ESTABLISHED	-
0	0 B	ACCEPT	tcp	*	*	0.0.0.0/0	0.0.0.0/0	tcp dpt:5000	AllowAdminService
0	0 B	zone_lan_forward	all	br-lan	*	0.0.0.0/0	0.0.0.0/0	-	-
0	0 B	reject	all	*	*	0.0.0.0/0	0.0.0.0/0	-	-

Chain *OUTPUT* (Policy: *ACCEPT*, 0 Packets, 0 B Traffic)

Pkts.	Traffic	Target	Prot.	In	Out	Source	Destination	Options	Comment
170	17.83 KB	ACCEPT	all	*	lo	0.0.0.0/0	0.0.0.0/0	-	-
1.46 K	644.26 KB	output_rule	all	*	*	0.0.0.0/0	0.0.0.0/0	-	Custom output rule chain

Figure 3 LuCI Web App UI – 1



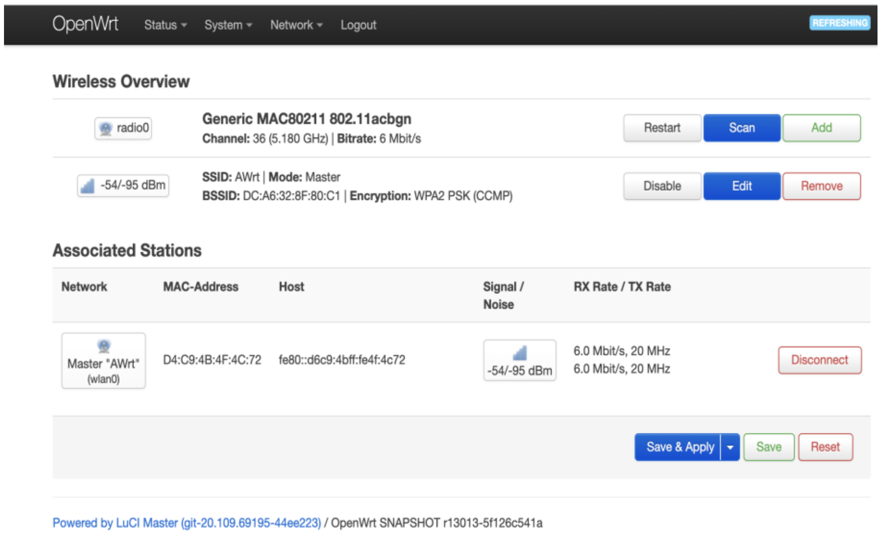
DHCP and DNS

Dnsmasq is a combined DHCP-Server and DNS-Forwarder for NAT firewalls

Server Settings

Domain required	✓ Don't forward DNS-Requests without DNS-Name
Authoritative	✓ This is the only DHCP in the local network
Local server	/lan/ — Local domain specification. Names matching this domain are never forwarded and are resolved from DHCP or hosts files only
Local domain	lan — Local domain suffix appended to DHCP names and hosts file entries
Log queries	☐ Write received DNS requests to syslog
DNS forwardings	/example.org/10.1.2.3 — List of DNS servers to forward requests to
Rebind protection	✓ Discard upstream RFC1918 responses
Allow localhost	✓ Allow upstream responses in the 127.0.0.0/8 range, e.g. for RBL services
Domain whitelist	ihost.netflix.com — List of domains to allow RFC1918 responses for
Local Service Only	✓ Limit DNS service to subnets interfaces on which we are serving DNS.
Non-wildcard	✓ Bind dynamically to interfaces rather than wildcard address (recommended as linux default)

Figure 4 LuCI Web App UI – 2



Wireless Overview

radio0	**Generic MAC80211 802.11acbgn** **Channel:** 36 (5.180 GHz) | **Bitrate:** 6 Mbit/s	
-54/-95 dBm	**SSID:** AWrt | **Mode:** Master **BSSID:** DC:A6:32:8F:80:C1 | **Encryption:** WPA2 PSK (CCMP)	

Associated Stations

Network	MAC-Address	Host	Signal / Noise	RX Rate / TX Rate
Master "AWrt" (wlan0)	D4:C9:4B:4F:4C:72	fe80::d6c9:4bff:fe4f:4c72	-54/-95 dBm	6.0 Mbit/s, 20 MHz 6.0 Mbit/s, 20 MHz

Powered by LuCI Master (git-20.109.69195-44ee223) / OpenWrt SNAPSHOT r13013-5f126c541a

Figure 5 LuCI 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.

Chapter 2. Project Architecture

Introduction

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

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

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

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

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.

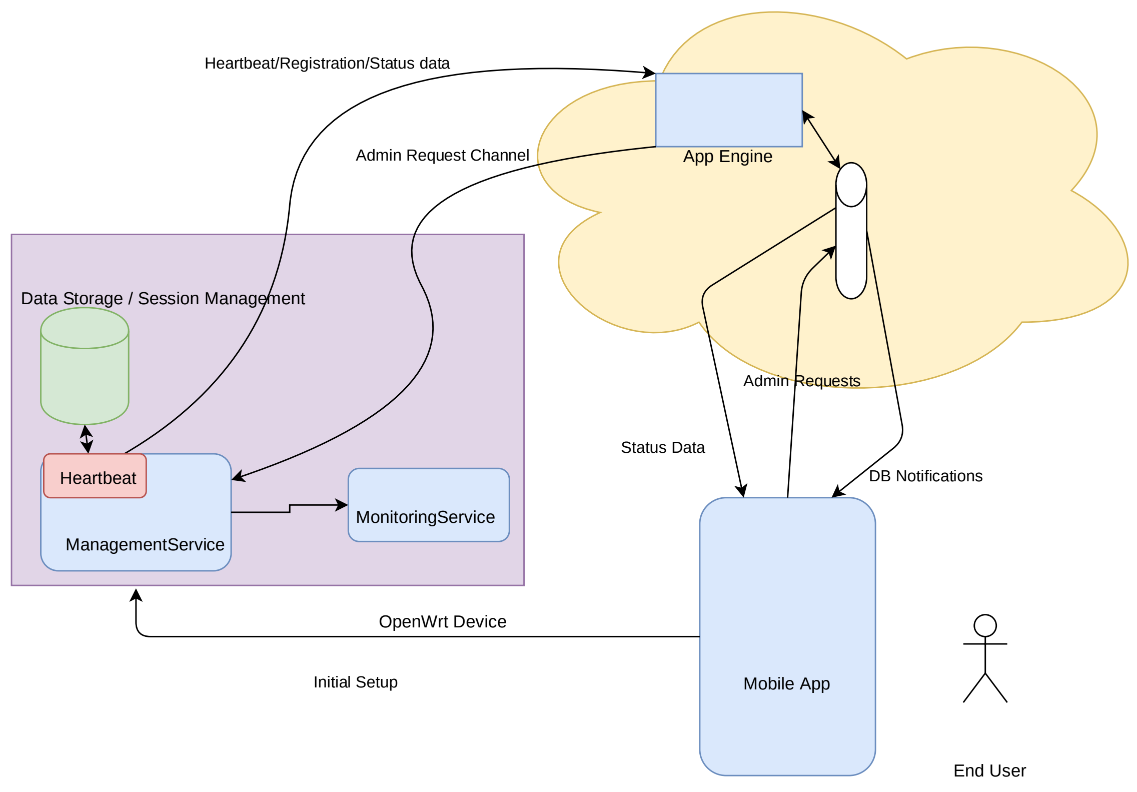


Figure 6 Project Architecture

Architecture Subsystems

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

The Mobile App used to provide the front-end functionality. The mobile app will provide authentication mechanism for connecting the cloud backend. Once the user is authenticated the user will be able to add or remove managed routers or manage an already added AmazeRT Router using the user interface provided.

The software that runs on the OpenWrt providing status updates and handling configuration changes is called the AmazeRT Agent. The AmazeRT Agent runs continuously on the router handling communication with the cloud backend for settings and status updates.

The backend logic to manage and handle communication between OpenWrt device and Mobile App runs on Google Cloud Platform. There are two subcomponents for this. One of them is to handle all the communication with the AmazeRT Agent. This component is named the AmazeRT App Engine. The other component is a cloud function which monitors changes in the database and send updates to the AmazeRT agent when the mobile app modifies a setting.

1) The Secure Shared Database keeps track of the device settings and provides real-time notifications to Mobile App about configuration changes. This also notifies the Cloud functions when a setting is updated by the mobile app, so that the same settings can be updated on the router side using the management request sent via Admin request channel.

Chapter 3. Technology Descriptions

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.

I. *Android SDK*

Android SDK is a comprehensive set of development tools for Android app development[10]. 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.

II. *JSch*

JSch is a Java implementation of SSH2 based on Java Cryptography Extension (JCE)[14]. 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.

III. GSon

Google GSon library serializes and de-serializes Java objects to JSON and back[15]. 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.

IV. Firebase Cloud Messaging

The Firebase Messaging service by Google provides APIs for accessing Firebase Realtime database [16]. 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 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.

I. 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 [17].

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.

II. 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]. AmazeRT 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

App Engine

App Engine is a platform as service infrastructure provided by Google to host an application without worrying about instance management, scaling, resource allocation etc [9]. 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 [10]. It is used to publish database modifications to App Engine websockets which in turn push the message to corresponding WRT device.

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 [8]. 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 [7]. MasterKey class references a key that is stored in Android Keystore. The recommended master key size is 256 bits. 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[7].

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. We use AES algorithm with 256 bit key in GCM mode for encryption[18].

Chapter 4. Project Design

Mobile App

The App flow diagram presented in Figure 7 gives an overall design of the AmazeRT Mobile app

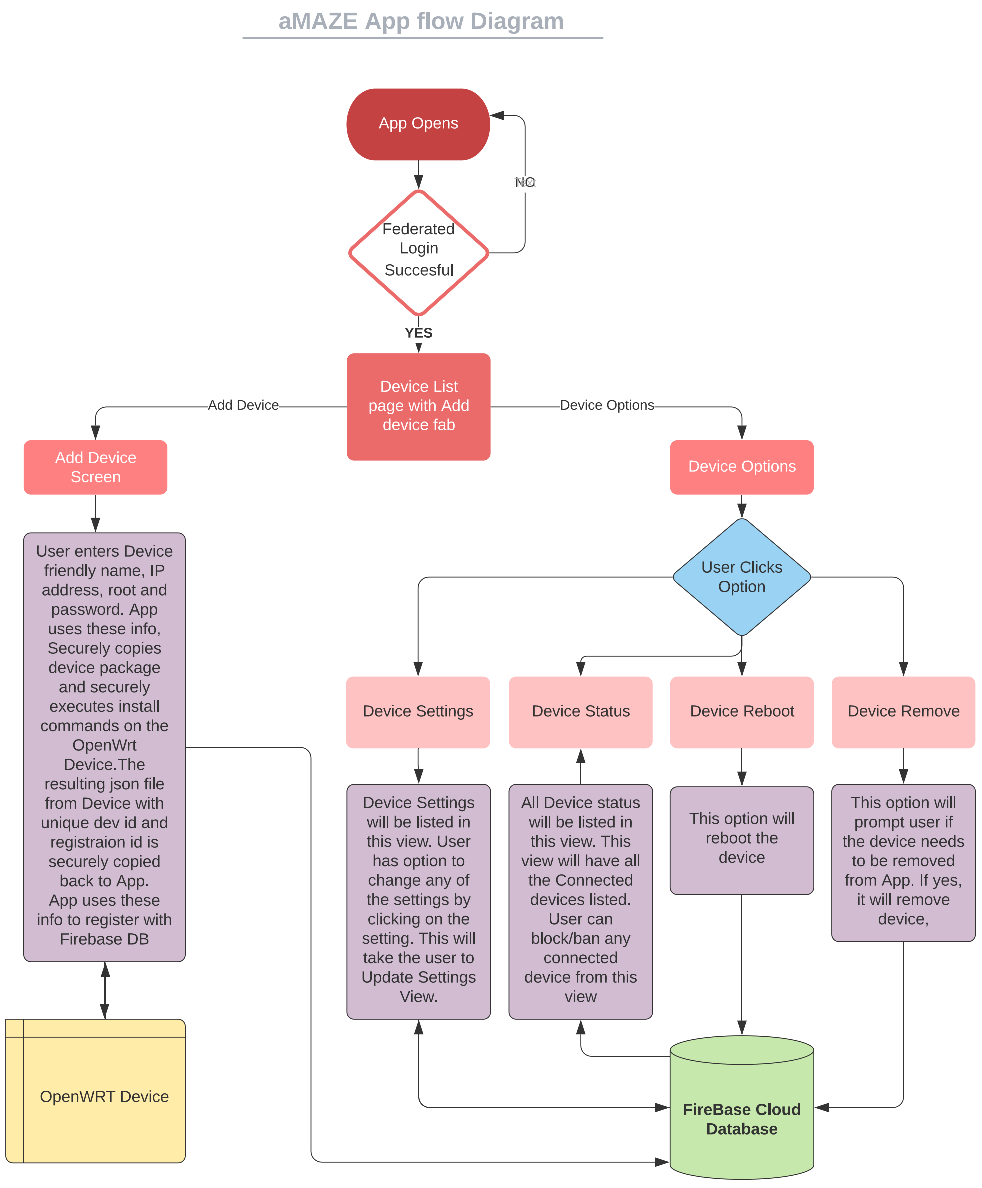

aMAZE App flow Diagram
App Opens
NO
Federated Login Succesful
YES
Device List page with Add device fab
Add Device
Device Options
Add Device Screen
Device Options
User Clicks Option
User enters Device friendly name, IP address, root and password. App uses these info, Securely copies device package and securely executes install commands on the OpenWrt Device.The resulting json file from Device with unique dev id and registraion id is securely copied back to App. App uses these info to register with Firebase DB
Device Settings
Device Status
Device Reboot
Device Remove
Device Settings will be listed in this view. User has option to change any of the settings by clicking on the setting. This will take the user to Update Settings View.
All Device status will be listed in this view. This view will have all the Connected devices listed. User can block/ban any connected device from this view
This option will reboot the device
This option will prompt user if the device needs to be removed from App. If yes, it will remove device,
OpenWRT Device
FireBase Cloud Database

Figure 7 App Flow Diagram

App Login UI

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

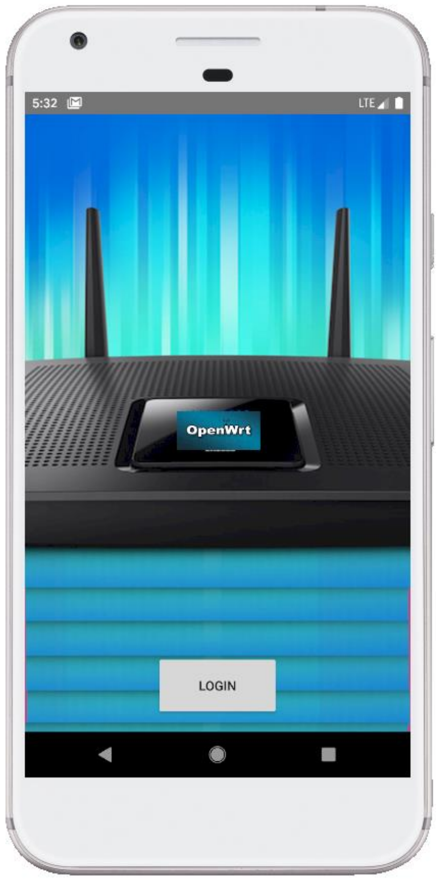


Figure 8 App Launch page

There 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.

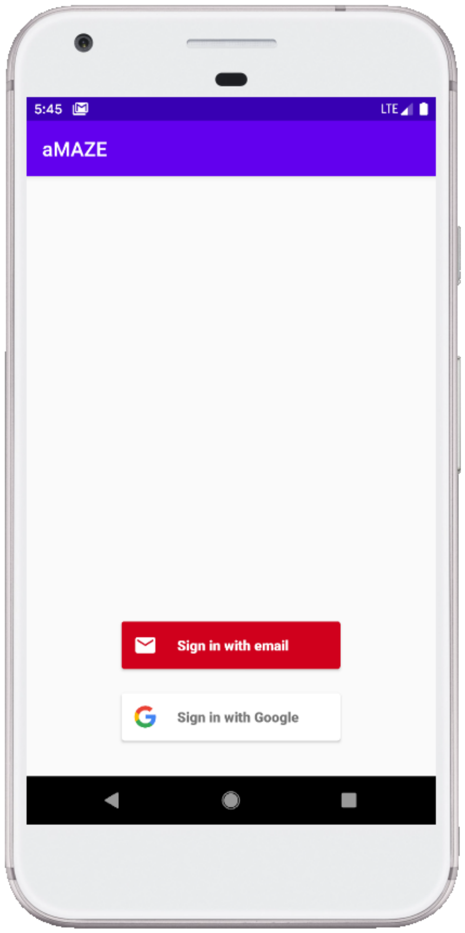


Figure 9 App Federated Login

Device Registration UI

Successful login will launch the Device list screen. This screen will list all the registered OpenWrt devices. 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.

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. Add 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 11. After the device registration is done, the Done button will appear on the same screen. Done button will take the user back to Device list screen. This screen lists the currently added device along with all OpenWrt devices registered before.

Figure 10 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, un-tar 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.

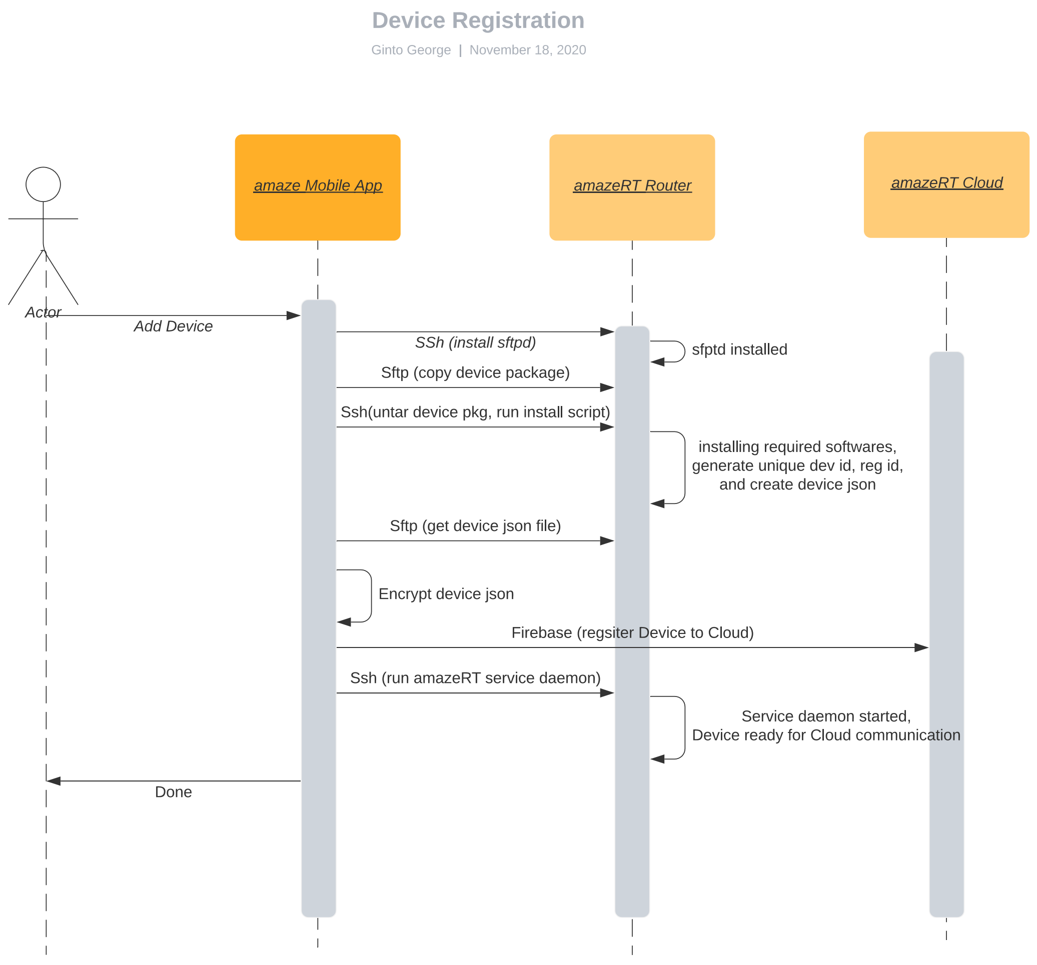

Device Registration
Ginto George | November 18, 2020
Actor
amaze Mobile App
amazeRT Router
amazeRT Cloud
Add Device
SSh (install sftpd)
sfptd installed
Sftp (copy device package)
Ssh(untar device pkg, run install script)
installing required softwares, generate unique dev id, reg id, and create device json
Sftp (get device json file)
Encrypt device json
Firebase (regsiter Device to Cloud)
Ssh (run amazeRT service daemon)
Service daemon started, Device ready for Cloud communication
Done

Figure 10 Device Registration Sequence Diagram

Secure Data Design

Settings data value read and written back to the Firebase Realtime Database can include critical data that needs to be secured. Symmetric Key Encryption is employed for this using a Symmetric Key shared between the Device and the Mobile App. A

SymKeyEncryption class is designed to encapsulate the encryption and decryption interfaces and other Encryption parameters. This class provides 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. APIs are designed to generate the Initialization Vector, to Encrypt input plaintext using Symmetric Key Encryption, to Decrypt input ciphertext using the shared Symmetric Key and Decode the padded cipher text to segregate the IV, Message Digest and Cipher text. SymKeyEncryption encapsulates data members for AES256_GCM secure key parameters of Key length, Iteration count and a random generated initialization vector to further strengthen the encryption.

Firebase Data Access handling

A FirebaseQueryLiveData class is used to add event listener for Firebase Realtime database data change events. LiveData Data holder class is extended by this class to allow for observing the changes in Database data snapshot. Design of the class is in a generic fashion to instantiate and query device settings or device status database paths.

DeviceSetting and Device Status ViewModel class instances use this class to interact with the Realtime Database.

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.

I. 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.

SymKeyEncryption class APIs are used to decrypt the encrypted data fetched from the Relatime database. FirebaseQueryLiveData class is used to implement the listener interface to the Realtime Database.

II. 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.

SymKeyEncryption class encrypt APIs are used to encrypt the settings data being pushed to the Realtime database.

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.

FirebaseQueryLiveData class is used to implement the listener interface to the Realtime Database.

User Profile UI

Mobile App has a Toolbar on all screens once logged in. The Toolbar has user profile icon on the right corner. Google profile photo is displayed on the user profile icon. Users click on user profile icon will take to 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 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 can 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 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 send 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 Infrastructure

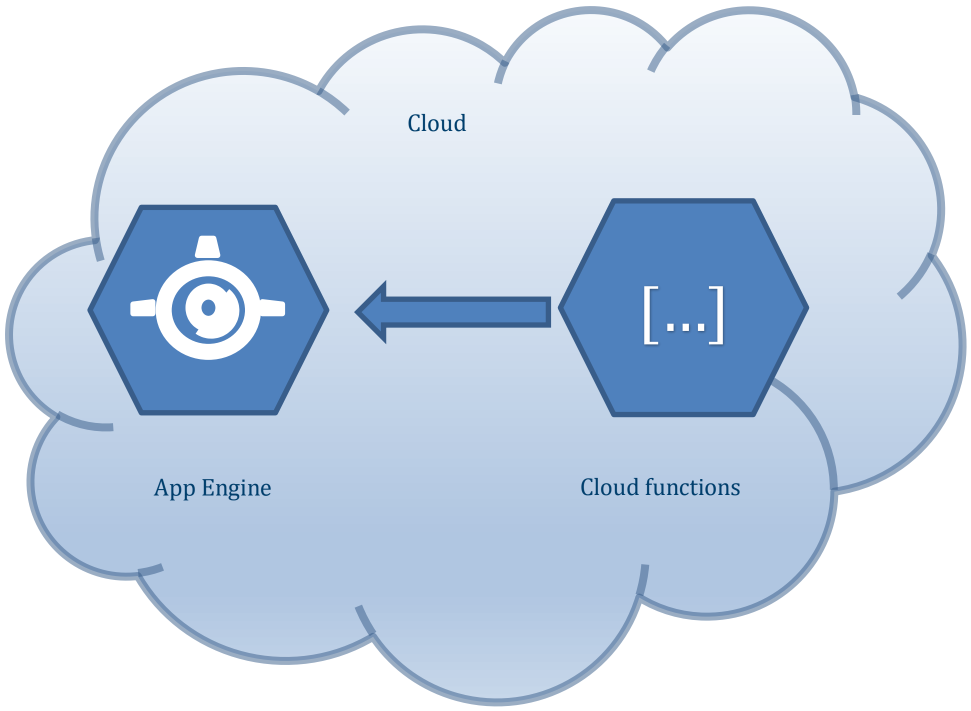


Figure 11 Cloud Architecture

The cloud infrastructure acts as the middle layer for the communication among the other components. It uses the google cloud components like AppEngine and Cloud Functions. The communication between these components are done using json messages send over web sockets.

AppEngine

AppEngine is implemented as a python Flask application running on a google AppEngine instance. When the AppEngine is fist initialized it create secure DB connection with Firebase real time database using the service account credentials. After that it generate a unique Id (to be used to identify that all the subsequent communications from DB to AppEngine) and update this unique id in Database. It starts the flask application which exposes two public websocket urls wss://amaze-id1.wl.r.appspot.com/registration and wss://amaze-id1.wl.r.appspot.com/notify for communication. The registration url is for the communication between AmazeRT agent and the AppEngine while the notify url is for the communication from DB.

Cloud functions

Google Realtime Database cloud function are serverless functions which get triggered for a particular event. In our project cloud function is used to send a message to AppEngine websocket whenever a data base change happens. The Cloud function is hosted in google cloud account which has both the Realtime data base and AppEngine instance in it. Cloud function also used private key and certificate to securely get authenticated to database for accessing the database fields for further creating the data required in the message to be send to the cloud AppEngine. Each message from cloud function should have the unique Id set by the AppEngine earlier in the DB. At each trigger of database change the cloud function reads the data base and get the key and add this to the message. This way cloud function makes sure that the notify message send by it in fact originated from cloud function itself.

Data Security

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.

I. *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.

Securing communication between device and Mobile App

OpenWrt routers have various settings that need to be controlled via the mobile app. Since the control and command requests to the device from the Mobile app are routed to the right device by the App Engine on the Cloud, we will need to secure the control and command channel so that no unauthorized person can issue control requests. To ensure this we have multiple layers of encryption added. All the communication between router and cloud, and between app and cloud happens with TLS encryption. This prevents an eavesdropper from gaining access to the device settings or controls.

The issue with this model is that, if a hacker somehow gains control of the cloud via leaked credentials, a rogue employee managing the cloud, or even a bug in the App engine, the security of the router can be compromised. This is especially important in case of sensitive data like Wifi password for the router. To handle this scenario, we do a second layer of encryption using a symmetric key shared just between the router and the mobile app. During the initial setup this key is created in the router and will be exchanged with the mobile app, using local ssh connection. This ensures that none other than the router and mobile app has access to this key. All the settings are then encrypted using this key, when sent across via cloud. At the moment, due to time constraints, the status parameters are not encrypted using this, but it is easily switchable to encrypted communication.

Chapter 5. 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. Each of the settings item captured in the recyclerview implements a on click listener to navigate to a Device Settings Update UI screen. Settings update is supported through this new update activity.

Device Settings Update UI

This UI component allows an authenticated user to update the OpenWrt device settings information. This component allows settings update for one setting at a time. It implements settings update data checks based on a settings support json that is fetched from the installed package. Conditional implementation of the UI textView / Switch is implemented based on the setting type. Apart from this, a help text is provided on detecting focus on the Edit text. This help text is also derived from the setting json file.

Once the user decides to apply the setting, the Firebase Realtime database settings reference is used to update the configuration settings database.

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 in Listing 1

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

Listing 1 - Structure of amazert.json

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 AmazeRT 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 format in Listing 2.

```
{
 {
    "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>
    }
 }
```

Listing 2 - Structure of Registration packet

The settings in the packet is always encrypted using the AES encryption with 256 bits 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, 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. The structure of the heartbeat packet is described in Listing 3

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

Listing 3 - Structure of Heartbeat packet

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 are 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

Cloud backend uses websocket over TLS also called secure websocket (wss) for the communication to AmazeRT agent as well as to the Firebase Realtime database cloud function. Python websocket libraries are used for the implementation.

AppEngine

AppEngine is implemented as a python Flask application running on a google AppEngine instance. When the AppEngine is fist initialized it create secure DB connection with Firebase real time database using the service account credentials. After that it generate a unique Id (to be used to identify that all the subsequent communications from DB to AppEngine) and update this unique id in Database. It starts the flask application which exposes two public websocket urls wss://amaze-id1.wl.r.appspot.com/registration and wss://amaze-id1.wl.r.appspot.com/notify for communication. The registration url is for the communication between AmazeRT agent and the AppEngine while the notify url is for the communication from DB.

I. *Registration url*

This url is used by all the OpenWrt devices running AmazeRT agent in it. Every message send to this interface has an *identifier* field in it which uniquely identify a device in the database. If this does not match with any entry in the data base this message is ignored. The AppEngine will never create or modify the identification field in the database. It is created and managed solely by the Amaze app. This way only if a device is registered with the database it can ever send get authenticated in the registration websocket interface. The devices use this url for updating its settings as well as status.

II. *Notify url*

This url is used only by the Realtime Database Cloud function. Whenever a change in data base data happens the cloud function is triggered. The cloud function sends a message to this url with the change in data and the unique key present in the DB. This unique key is used as a cookie to verify that the message is originated from the data base.

III. *Messages*

Following messages are used for the communication by the Cloud back end other components.

Sample notification message from Database to AppEngine is Listing 4

```
{
  "key":"1739292a-1f32-11eb-a51d-0242ac110004",
 "resource_string":"projects/_/instances/amaze-
id1/refs/users/_SDFsEfRSDjFCZASEf/532e8c40-18cd-11eb-a4ca-dca6328f80c0",
   "data":{"settings": {"0": {"value": "Wrt1"}, "1": {"value": " PiWRT"}}}
}
```

Listing 4 - Sample notification from Database to App Engine

Sample notification message from AppEngine to Device is in Listing 5

```
{
    "identifier" : {
      "email": "nabin@gmail.com",
      "uid": "_SDFsEfRSDjFCZXCVASEf",
      "deviceId": "fb967061-168a-11eb-9272-88e9fe6b97d6"
    },
    "action": "setting",
    "setting" : { "name" : "wireless.wifinet0.ssid", "value" : "PiWRT" }
}
```

Listing 5 - Sample notification from App Engine to AmazeRT Agent

Cloud functions

Google Realtime Database cloud function are serverless functions which get triggered for a particular event. In our project cloud function is used to send a message to AppEngine websocket whenever a data base change happens. The Cloud function is hosted in google cloud account which has both the Realtime data base and AppEngine instance in it. Cloud function also used private key and certificate to securely get authenticated to database for accessing the database fields for further creating the data required in the message to be send to the cloud AppEngine. Each message from cloud function should have the unique Id set by the AppEngine earlier in the DB. At each trigger of database change the cloud function reads the data base and get the key and add this to the message. This way cloud function makes sure that the notify message send by it in fact originated from cloud function itself.

Data Security

Shared Configuration Database

Google Firebase Realtime database is used for storing the shared Settings and configuration data of all amazeRt devices. Firebase Realtime database rules ensure data access control. To ensure data privacy of each of the managed device, the primary rule limits the read and write access to individual device data to the corresponding firebase authenticated user. Sample set of rules used is in Listing 6

```
{
  "rules": {
    "$uid": {
      ".write": "$uid === auth.uid",
      ".read": "true",
      "device": {
        "$deviceuid":{
          "Wifi":{
            ".validate": "newData.isString() &&
newData.val().length < 4 && ((newData.val().toUpperCase()
=== 'ON') || (newData.val().toUpperCase() === 'OFF'))",
            ".read": "true"
          },
          "PowerState":{
            ".validate": "newData.isString() &&
newData.val().length < 4 && ((newData.val().toUpperCase()
=== 'ON') || (newData.val().toUpperCase() === 'OFF'))",
            ".read": "true"
```

```
                         }
                    }
               }
          }
     }
}
```

Listing 6 - Rules for Database Access Control

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 bits. The key encryption scheme AES256_GCM_SPEC is used. The same key is used to decrypt the file for App use

Securing communication between device and Mobile App

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 Figure 12 below and stored in the Firebase Realtime database.

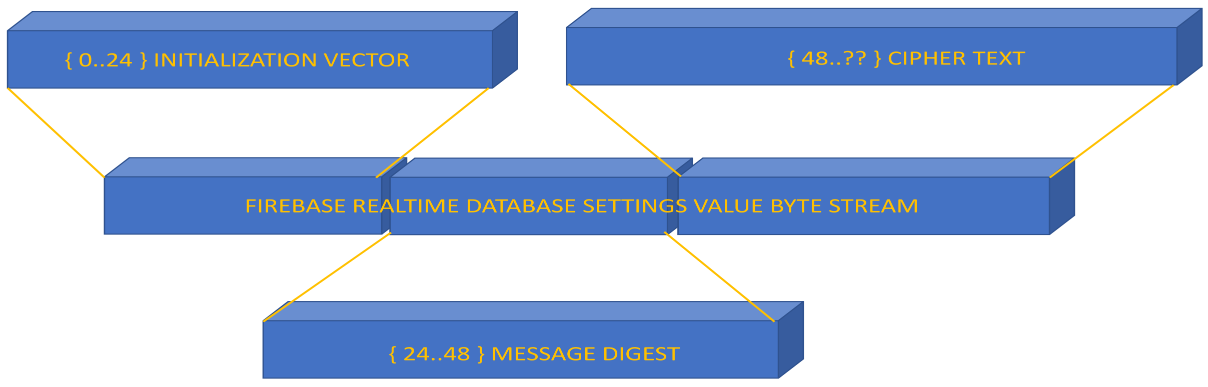

{ 0..24 } INITIALIZATION VECTOR
{ 48..?? } CIPHER TEXT
FIREBASE REALTIME DATABASE SETTINGS VALUE BYTE STREAM
{ 24..48 } MESSAGE DIGEST

Figure 12 Packing IV|MessageDigest|CipherText

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. The API level chosen for Android device is API 29.

Chapter 6. Testing and Verification

Testing Process

Mobile App

I. 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.

II. 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

I. 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.

II. 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.

III. 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.

IV. 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.

V. 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

A dummy html page was hosted along with the cloud to test the cloud infrastructure. This page opens a WebSocket connection with the cloud App Engine and has the capability to send and receive json to and from the cloud. The page provides one text box to simulate the registration command (the registration/heartbeat command send by the device) and another text box to simulate the notify command send by the database change through cloud function. User can enter the Jason as text in these test boxes and test the cloud framework. The reply from the cloud will be printed on the page. Using multiple instances of the page user can test the scenario of multiple AmazeRT devices.

I. *Fake device registration / Heartbeat messages*

The Cloud infrastructure is supposed to ignore any unauthorized messages send to the WebSocket. Websocket is protected by TLS encryption. If the use rid or device id in the message send to it is not matching with the value present the data base the message is ignored.

II. *Fake Data base notification*

The notification message send from the database cloud function should also be authorized. If the unique id registered by the App Engine in the data base is not present in the notification message the message is discarded.

Test results

AmazeRT Agent

I. *Registration packet generation*

The registration packet generated from AmazeRT Agent is shown in

```
{
    "action": "register",
    "settings": [{
        "name": "system.@system[0].hostname",
        "value":
"2PNPgY1oR/meBxgKDqpXCw==m+qU4b2keBIRbg/yyNq9Yg==s7cT2tFF3
g=="
    }, {
        "name": "wireless.wifinet0.ssid",
        "value":
"GXrlru+6Y3tAVR3HhDuF7A==axRWkgdMYDWfxcZZuclUVg==dcSdAg=="
    }, {
        "name": "wireless.radio0.country",
        "value":
"9zNf8rSzaK3e3+1xV821Og==Bs8w7C4PDREEBlvlbD24Mg==0ho="
    }, {
        "name": "wireless.wifinet0.macfilter",
        "value":
"ju3SOZneUDRpcLPbeCg0Vw==fxUBfORKNuR5Me8rLaKZEg==F7dW/9qiA
A=="
    }, {
        "name": "wireless.wifinet0.maclist",
        "value":
"xCHHCJGD1u3SIr2M1jLLJw==qiPay5WE9jG8sp0UiW8YsA=="
    }, {
        "name": "wireless.wifinet0.disabled",
        "value":
"L4LBCneE19hTQlTJ1MUasg==CdqFNqCvmwr12Vyiaf3XPw==bA=="
    }],
    "status": [{
        "name": "wifi.clients",
        "value": "Command failed: Not found"
    }, {
        "name": "amazert.heartbeat.time",
        "value": "Mon Nov 30 01:04:03 UTC 2020"
    }, {
        "name": "dhcp.leases",
        "value": {
            "dhcp_leases": [],
            "dhcp6_leases": []
        }
    }, {
        "name": "assoclist.wlan0",
        "value": {
            "results": []
        }
    }, {
        "name": "amazert.poweron.time",
        "value": "Mon Nov 30 01:04:03 UTC 2020"
    }, {
        "name": "amazert.status",
        "value": "running"
    }],
    "identifier": {
        "uid": "0NWFFJG764T2ucPAhdLDNwQjcgA2",
        "deviceId": "f0524c82-2614-11eb-b140-
dca6328f80c0",
        "email": "nabin.thomas@gmail.com"
    }
}
```

Listing 7 - Test Results showing registration packet

II. Changing a setting

The terminal output in Listing 8 shows the outputs of hostname value before and after a setting change. The corresponding logs from AmazeRT Agent log file is shown in Listing 9

```
root@AmazeRT:~# uci show system.@system[0].hostname
system.cfg01e48a.hostname='AmazeRT'
root@AmazeRT:~# uci show system.@system[0].hostname

system.cfg01e48a.hostname='AmazeRT-Nabin'
```

Listing 8 - Terminal output showing before and after a system setting change

```
root@AmazeRT:~# uci show system.@system[0].hostname
system.cfg01e48a.hostname='AmazeRT'
root@AmazeRT:~# uci show system.@system[0].hostname

system.cfg01e48a.hostname='AmazeRT-Nabin'
```

Listing 9 - Logs showing a system setting change

Mobile App

I. Device Registration UI

Device List screen with fab button to add device. No device listed before any device registration. (Figure 13)

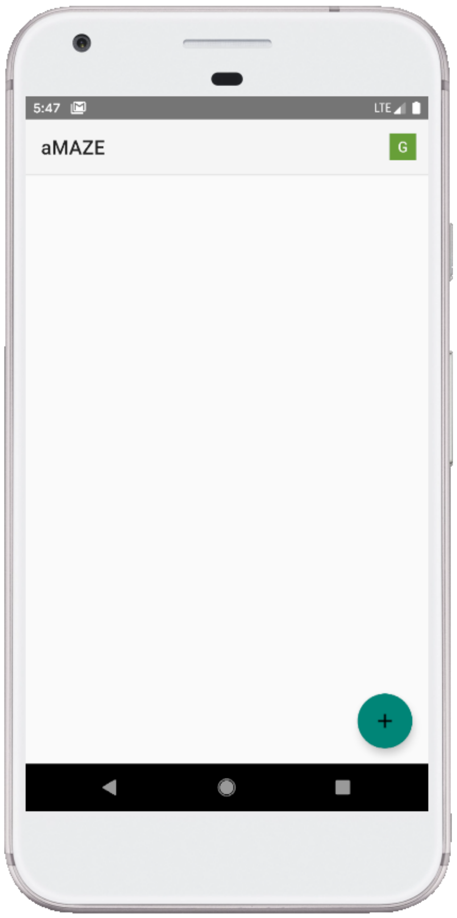
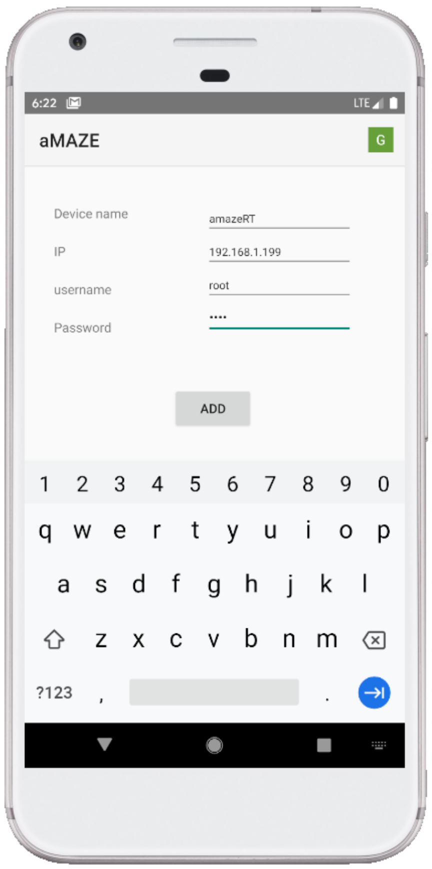


Figure 13 Device List screen with fab button to add device

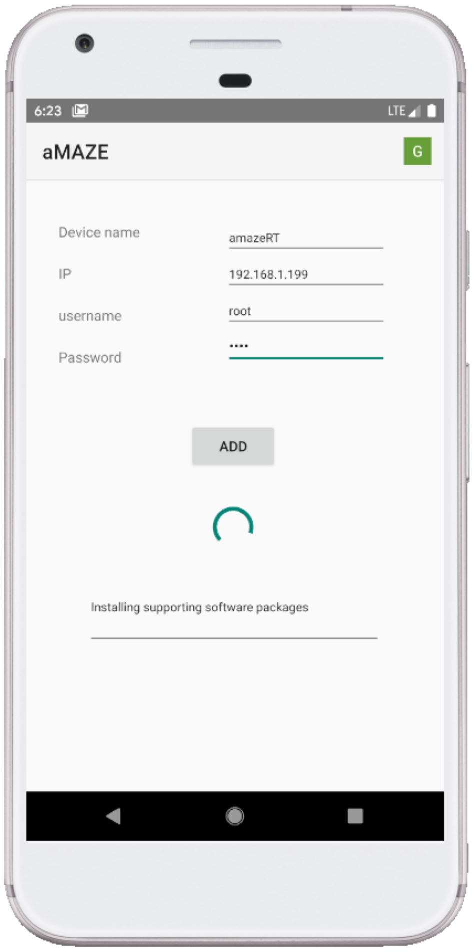
Add device screen with mandatory fields for friendly device name, IP address, hostname, and password. (Figure 14)



Device name	amazeRT
IP	192.168.1.199
username	root
Password	••••

Figure 14 App, Add device for registration

Add device screen with registration circular progress bar and short text showing the current registration updates (Figure 15)



Device name amazeRT

IP 192.168.1.199

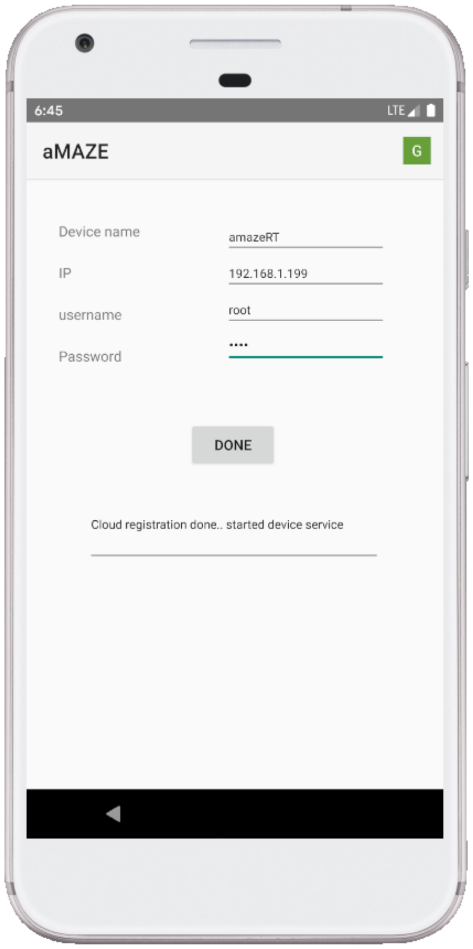
username root

Password ••••

Installing supporting software packages

Figure 15 App, Device Registration Progress

Add device screen with registration done. Done button will take user back to Device list screen (Figure 16)



Device name amazeRT

IP 192.168.1.199

username root

Password ••••

Cloud registration done.. started device service

Figure 16 App, Device registration Done

Device List Screen with newly added device listed in the recycler view. (Figure 17)

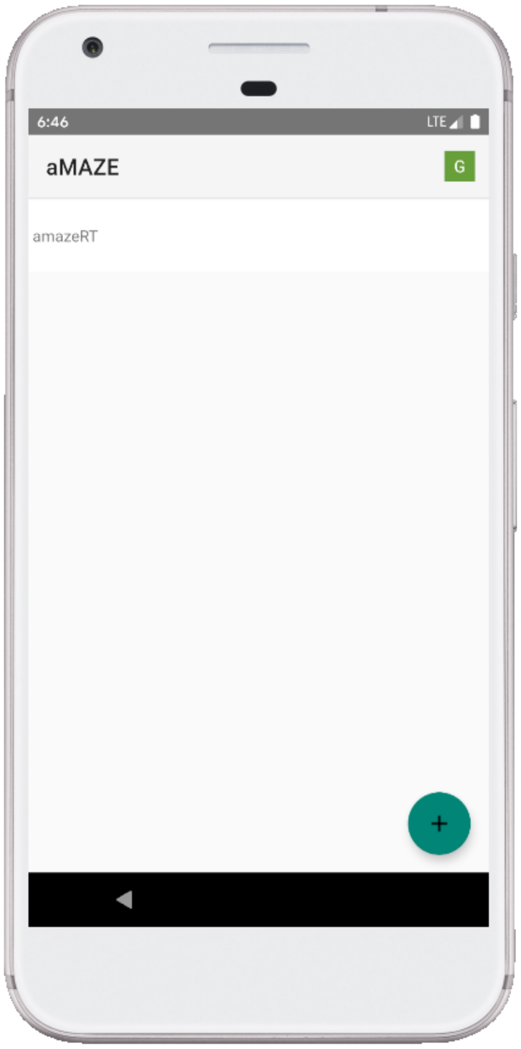
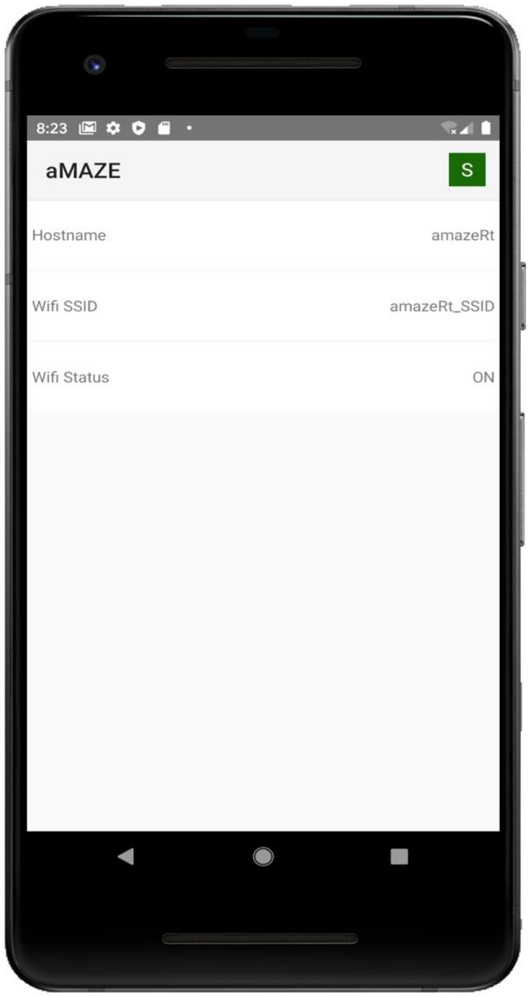


Figure 17 App, Device List with newly added device

II. Device Settings UI

Device Settings screen listing the settings from device. (Figure 18)



Hostname amazeRt

Wifi SSID amazeRt_SSID

Wifi Status ON

Figure 18 App, Device Settings Screen

Update device screen on clicking any of the settings on Device Settings screen. User can change the settings. (Figure 19)

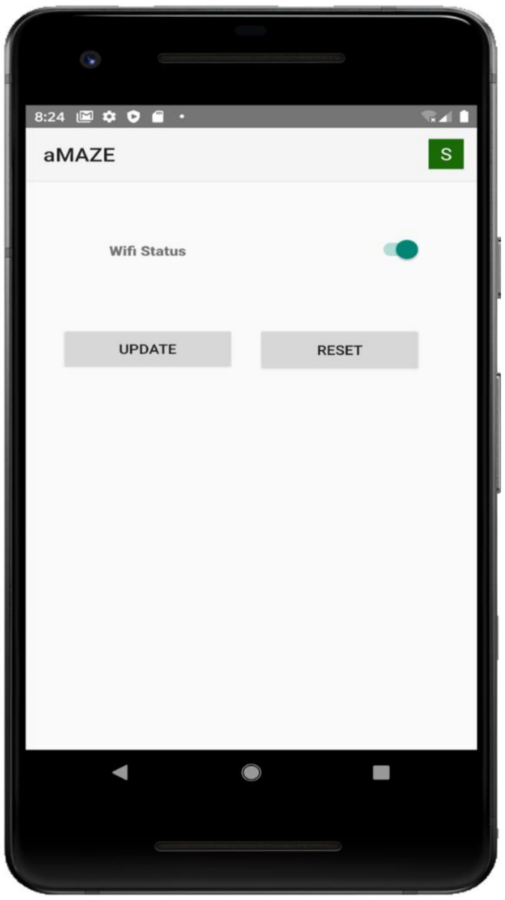
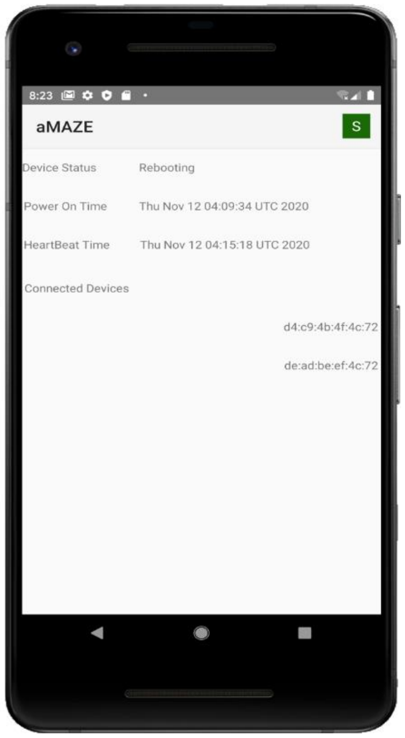


Figure 19 App, Device Settings Update Screen

III. Device Status UI

Device status screen with all the status from the device. User can block or ban a connected device from this screen. (Figure 20)


8:23

aMAZE

S

Device Status	Rebooting
Power On Time	Thu Nov 12 04:09:34 UTC 2020
HeartBeat Time	Thu Nov 12 04:15:18 UTC 2020
Connected Devices	
	d4:c9:4b:4f:4c:72
	de:ad:be:ef:4c:72

Figure 20 App, Device Status Screen

IV. User Profile UI

User Profile screen shows the user details including user Profile picture. (Figure 21). Logout button will logout user out of the Google federated login. This will take user back to launch screen.

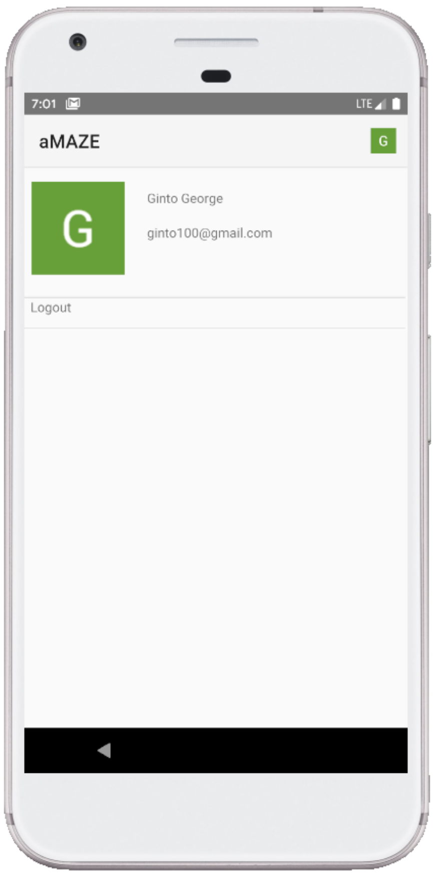
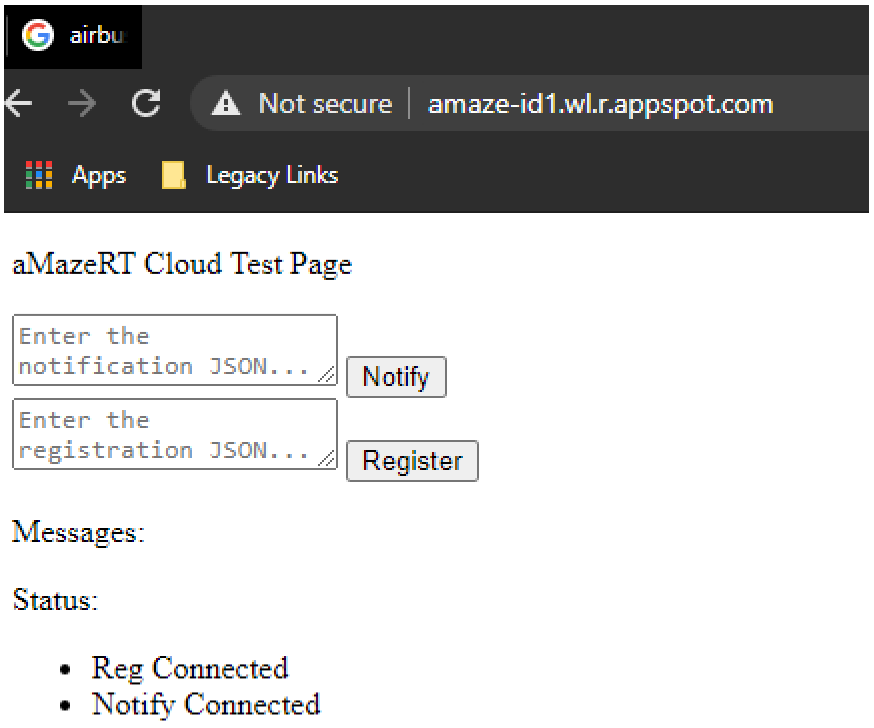


Figure 21 App, User Profile screen

Cloud Backend

A Test Page to test the AppEngine and Cloud Function was created. The Register message can be used to send a json message to the cloud to simulate the registration of a device. The notify message is used to send a cloud function notification message to the AppEngine , which will be forwarded to the AmazeRT Agent (Figure 22).



aMazeRT Cloud Test Page

Enter the notification JSON...

Enter the registration JSON...

Messages:

Status:

- Reg Connected
- Notify Connected

Figure 22 Cloud Testing framework

Chapter 7. Performance and Benchmarks

Scalability

Google Cloud provides seamless scalability for applications using it. App Engine and Cloud functions are a scalable system which automatically add more capacity as workloads increase. Googles proven cloud infrastructure provides enterprise level scaling features. Even though the design and infrastructure are scalable the billing account needs to be set up accordingly to make use of this feature. For this project as of now the minimal scaling provided by google is used.

Throughput

Cloud infrastructure is the major bottleneck for the throughput of the whole system. The cloud function AppEngine communication is the single place where most of the traffic happens. Google provides up to 10000 instances of AppEngine instances per day. Which is way above the requirement for this project. Another area is the Firebase Realtime Database updates. As per Google 200,000 simultaneous connections are supported, and 100,000 simultaneous responses can be handled. That means up to 200,000 mobile Applications can connect at the same time and out of that half of them can be updating the DB simultaneously. The design of the data base is such that the change from a device-mobile app pair is localized in the database. This helps improve the performance.

Reliability

The entire AmazeRT management system depends on the reliability of the four major components in the system namely the AmazeRT Agent, Cloud backend, Cloud database, and the Android app. For the cloud backend and database, once we handle the runtime errors due to design and coding issues correctly, the reliability of the infrastructure mainly depends on the cloud service providers. Since we are using the free or trial tiers for these, performance benchmark is not added as the criteria for those components.

For the AmazeRT agent, we need it to be lightweight enough to not slow down the router too much, while keeping the settings updated in the cloud without much delay. For the prototype purposes, we chose the criteria that the data should not be out of date for more than 30 seconds at any time. Even if the agent crashes, the recovery mechanism should be able to update the settings within 30 seconds of the timeframe. To handle this, we used a 10 second timer for our prototyping tests and was able to achieve the target.

For the Mobile app side, our target was to not have any app crashes or noticeable delays in the user interface while the app is being used. The end to end testing we did not find any unexpected issues there.

Feature Parity

Although our original goal was to implement the management system at feature parity to LUCI Web client for OpenWrt, that was a huge project considering the number of features that LUCI supports. Our trimmed down goal was to support a minimal set of features for end to end flow but providing a cloud-based management that is not supported by LUCI. Such systems are available for commercially available consumer routers, but none of that exist for OpenWrt. We were able to implement such a system with a framework for adding more settings. Although we don’t support all the features in LUCI or commercial routers, eventually AmazeRT will be able to implement support for every parameter configurable via LUCI.

Chapter 8. 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. Cloud functions are auto scalable. Google instantiates Cloud functions depending on the traffic.

App Engine

App Engine is deployed in google cloud as a compute instance. It provides a public URL and a persistent websockets based connection to the managed devices. Whenever the App Engine application is upgraded, the Google infrastructure makes sure that the service is continuous and seamless. This avoid any App Engine downtime.

Android App and Device software

Android App is deployed on mobile. The app is used for managing and monitoring the device. Android App is currently deployed as a sideloaded apk. The future plan is to deploy this as an app in Google Playstore.

Packaging AmazeRT Agent

As part of the build process for AmazeRT Agent, the scripts are compressed together as tar file, and is checked in as a resource file in the Android App's repository. The App may then use that resource when it is setting up the router, to copy the tar file to the router and run the installation process.

Secure Database operations

Firebase Realtime Database

Firebase RealTime database is used for the AmazeRT device to Mobile App communication. Firebase Real time database is hosted/deployed in cloud and is well covered by Google's support and service offerings. Google Firebase console also provides a consolidated access point for all database and rules control and configuration changes. Firebase Real time database also comes with a full-fledged API to support database management.

AmazeRT Agent data storage

AmazeRT agent relies on the initial setup data to be persistent. This data identifies the device uniquely to the cloud backend. It also helps to generate key for encrypting communication with the mobile app. The registration data is saved as a JSON Object in the location /etc/amazert.json, in line with other Linux based applications that save configuration information. These locations are protected by the default access control mechanisms in the Linux operating system and hence cannot be read by anyone unless they already have access to the router. In addition to this, the agent keeps its own runtime data, such as the encryption key and the device settings' local cache. These information are saved in the process' address space and are protected from attacks by the Linux operating system. In this case also we are assuming that we don't need to protect the data from root user, since protecting that in a user mode application will be almost impossible without have kernel components for this purpose. However, this is not a showstopper for the project, as any such system the root access may be restricted to the owner and only from the local network in case of OpenWrt by default.

Github repository

Git was chosen as the version control software for its ease of use, and good integration with development tools across Android, Google Cloud and Visual Studio code.

For the purpose of creating a central repository that can be shared across everyone, github platform was chosen.

All the project artifacts are saved version controlled in the github repository located at https://github.com/nabinthomas/amazert

The repository is organized to the following sections (as directories) to separate different modules and minimize the conflicts due to parallel development.

1- Learn – Contains our experimental code that we used for learning and research purposes.

2- aMAZE – Contains the source for the Mobile app.

3- amazert – Contains the source for the AmazeRT Agent which runs on the OpenWrt router.

4- AppEngine – Contains all the sources used to run the Cloud backend’s App Engine part.

5- CloudFunctions – Contains the sources used to handle events from the cloud database when data changes.

Chapter 9. Summary, Conclusions, and Recommendations

Summary

When the project was initially conceived, it was intended 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 most of the agent and cloud backend implementations can be reused for implementing such a solution.

While the AmazeRT management system is not able to manage all the configurations of the OpenWrt device yet, it is a step in the right direction for providing support for Mobile app and cloud-based configuration management.

Conclusions

Our original intent with the project was to provide a remote cloud-based management solution for OpenWrt routers, like what is provided with commercial consumer routers. However, the timeframe we had prevented us from implementing full support. We were able to implement the entire framework with limited set of settings for the router. Since the implementation was done in an extensible way, more settings can be added with minimal effort. With our effort to implement the management system for OpenWrt based routers, we spent considerable amount of time studying the various aspects of controlling devices remotely in a secure way. The learnings from those helped us understand that the architecture that we proposed can be implemented in a variety of scenarios for managing devices remotely. With the design we implemented we made it such that this can be ported to other devices with minimal effort. Given the current rise in popularity of EDGE Computing and IOT devices, this architecture can be used to implement management solutions for those devices. Many of such devices may be having a direct connection to them at the time of installation and setup but may have internet connectivity to communicate with cloud backends.

Secure communication from cloud to device and from cloud to application was an important requirement addressed in the development of the project. Application and the device share a symmetric private key agreed upon on the initial setup of the device which is used to encrypt all the messages communicated between the app and the device later. Also, all the communication to and from the cloud is TSL encrypted. This secure architecture can be used by any framework implementing similar communication mechanism. Given the robustness of the security aspects we have implemented in the project it is safer to deploy this for a commercial or business environment than the current web-based router management interface.

Recommendations for Further Research

With the research and debug done for implementation of the AmazeRT software stack, a number of potential enhancements for this has been identified such as

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 the Mobile app client 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.
- Adding support for the entire settings for OpenWrt routers
- Adding support for remote monitoring of more parameters for the OpenWrt Router.
- Enhancing the infrastructure to support device management for Edge computing devices.
- Integrating this entire software suite with the open source OpenWrt codebase and making this a default install option for all OpenWrt devices.
- Publishing the production version of the App on Google Play store and deploying the production versions of the cloud backend code for public use.
- Implementing a framework for continuous integration and testing on a cloud-based CI/CD platform like CircleCI.
- Publishing the source code as open source so that the software design can evolve with community support to be in sync with the latest updates on OpenWrt, Cloud platforms and Mobile app platforms.

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.
UI	User Interface
CI/CD	Continuous integration and continuous delivery
IoT	Internet of Things
SDK	Software Development Kit
IP	Internet Protocol
DB	Database
SSH	Secure Shell
AES	Advanced Encryption Standard
GCM	Galois/Counter Mode

continued

Acronym	Description
HKDF	Hash-based Key Derivation Function
HW	Hardware
RAM	Random Access Memory
TLS	Transport Layer Security
CPU	Central Processing Unit
UID	Unique Identifier
URL	Uniform Resource Locator
IV	Initialization Vector
PBKDF2	Password-Based Key Derivation Function 2
OS	Operating System
JCE	Java Cryptography Extension
AWS	Amazon Web Services

Table 1 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]

[7] "Cryptography :   Android Developers," Android Developers. [Online]. Available: https://developer.android.com/guide/topics/security/cryptography. [Accessed: 06-Dec-2020].

[8] "Add Firebase to your Android project," Google. [Online]. Available: https://firebase.google.com/docs/android. [Accessed: 06-Dec-2020].

[9] "App Engine Application Platform | Google Cloud," Google. [Online]. Available: https://cloud.google.com/appengine. [Accessed: 06-Dec-2020].

[10] Android Developers. [Online]. Available: https://developer.android.com/. [Accessed: 06-Dec-2020]

[11] "Your First Function: Python | Cloud Functions Documentation," Google. [Online]. Available: https://cloud.google.com/functions/docs/first-python. [Accessed: 06-Dec-2020].

[12] "OpenWrt", Wikipedia, 28-Nov-2020. [Online]. Available: https://en.wikipedia.org/wiki/OpenWrt. [Accessed: 06-Dec-2020].

[13] K. Singh, P, "15 Best Apps That Can Help You To Control Your Router (2020 Edition)," TechViral, 25-May-2020. [Online]. Available: https://techviral.net/best-apps-that-can-help-you-to-control-your-router/. [Accessed: 06-Dec-2020].

[14] "JCraft," JSch - Java Secure Channel. [Online]. Available: http://www.jcraft.com/jsch/. [Accessed: 06-Dec-2020].

[15] Google, "google/gson," GitHub. [Online]. Available: https://github.com/google/gson. [Accessed: 06-Dec-2020].

[16] "Firebase Cloud Messaging," Google. [Online]. Available: https://firebase.google.com/docs/cloud-messaging/. [Accessed: 06-Dec-2020].

[17] "Getting started" Getting started - websockets 8.1 documentation. [Online]. Available: https://websockets.readthedocs.io/en/stable/intro.html. [Accessed: 06-Dec-2020].

[18] L. Wagner, "AES-256 Cipher – Python Cryptography Examples," DEV Community, 07-Feb-2020. [Online]. Available: https://dev.to/wagslane/aes-256-cipher-python-cryptography-examples-10b2. [Accessed: 06-Dec-2020].

Appendices

Appendix A – Rules for handing settings

```
dataDrivenSettingsRules = [ {
    "name" : "system.@system[0].hostname",
    "handler" : {
        "read" : {
            "commandType" : "uci"
        },
        "write" : {
            "commandType" : "uci"
        }
    }
}, {
    "name" : "wireless.wifinet0.ssid",
    "handler" : {
        "read" : {
            "commandType" : "uci"
        },
        "write" : {
            "commandType" : "uci",
            "epilogue" : ["wifi"]
        }
    }
}, {
    "name" : "wireless.radio0.country",
    "handler" : {
        "read" : {
            "commandType" : "uci"
        },
        "write" : {
            "commandType" : "uci",
            "filter" :
["AF","AX","AL","DZ","AS","AD","AO","AI","AQ","AG","AR","AM","A
W","AU","AT","AZ","BS","BH","BD","BB","BY","BE","BZ","BJ","BM",
"BT","BO","BQ","BA","BW","BV","BR","IO","VG","BN","BG","BF","BI
","KH","CM","CA","CV","KY","CF","TD","CL","CN","CX","CC","CO","
KM","CK","CR","HR","CU","CW","CY","CZ","CD","DK","DJ","DM","DO"
,"TL","EC","EG","SV","GQ","ER","EE","ET","FK","FO","FJ","FI","F
R","GF","PF","TF","GA","GM","GE","DE","GH","GI","GR","GL","GD",
"GP","GU","GT","GG","GN","GW","GY","HT","HM","HN","HK","HU","IS
","IN","ID","IR","IQ","IE","IM","IL","IT","CI","JM","JP","JE","
JO","KZ","KE","KI","XK","KW","KG","LA","LV","LB","LS","LR","LY"
,"LI","LT","LU","MO","MK","MG","MW","MY","MV","ML","MT","MH","M
Q","MR","MU","YT","MX","FM","MD","MC","MN","ME","MS","MA","MZ",
"MM","NA","NR","NP","NL","AN","NC","NZ","NI","NE","NG","NU","NF
```

```
","KP","MP","NO","OM","PK","PW","PS","PA","PG","PY","PE","PH","
PN","PL","PT","PR","QA","CG","RE","RO","RU","RW","BL","SH","KN"
,"LC","MF","PM","VC","WS","SM","ST","SA","SN","RS","CS","SC","S
L","SG","SX","SK","SI","SB","SO","ZA","GS","KR","SS","ES","LK",
"SD","SR","SJ","SZ","SE","CH","SY","TW","TJ","TZ","TH","TG","TK
","TO","TT","TN","TR","TM","TC","TV","VI","UG","UA","AE","GB","
US","UM","UY","UZ","VU","VA","VE","VN","WF","EH","YE","ZM","ZW"
],
            "epilogue" : ["wifi"]
        }
    }
}, {
    "name" : "wireless.wifinet0.macfilter",
    "handler" : {
        "read" : {
            "commandType" : "uci",
            "default" : "disable"
        },
        "write" : {
            "commandType" : "uci",
            "filter" : ["allow", "deny", "disable"],
            "epilogue" : ["wifi"]
        }
    }
}, {
    "name" : "wireless.wifinet0.maclist",
    "handler" : {
        "read" : {
            "commandType" : "uci",
            "default" : ""
        },
        "write" : {
            "commandType" : "uci",
            "epilogue" : ["wifi"]
        }
    }
}, {
    "name" : "wireless.wifinet0.disabled",
    "handler" : {
        "read" : {
            "commandType" : "uci",
            "default" : "0"
        },
        "write" : {
            "commandType" : "uci.custom",
            "filter" : {
                "0" : ["uci", "delete",
"wireless.wifinet0.disabled"],
                "1" : ["uci", "set",
"wireless.wifinet0.disabled=1"]
            },
            "epilogue" : ["wifi"]
        }
    }
}]
```

Appendix B – Rules for reading status

```
[{
    "name": "wifi.clients",
    "command" : ["ubus", "call", "hostapd.wlan0",
"get_clients"]
}, {
    "name": "amazert.heartbeat.time",
    "command" : ["date", "-u"]
}, {
    "name": "dhcp.leases",
    "command" : ["sh", "/usr/bin/amazert/dhcpclients.sh"]
}, {
    "name": "assoclist.wlan0",
    "command" : ["/usr/bin/python3",
"/usr/bin/amazert/assoclist.py", "wlan0"]
}, {
    "name": "amazert.poweron.time" ,
    "command" : ["cat", "/var/log/amazert.start"]
}, {
    "name": "amazert.status" ,
    "command" : ["echo", "running"]
}]
```

Appendix C – Sample registration packet

```
{
    "action": "register",
    "settings": [{
        "name": "system.@system[0].hostname",
        "value":
"2PNPgY1oR/meBxgKDqpXCw==m+qU4b2keBIRbg/yyNq9Yg==s7cT2tF
F3g=="
    }, {
        "name": "wireless.wifinet0.ssid",
        "value":
"GXrlru+6Y3tAVR3HhDuF7A==axRWkgdMYDWfxcZZuclUVg==dcSdAg=
="
    }, {
        "name": "wireless.radio0.country",
        "value":
"9zNf8rSzaK3e3+1xV821Og==Bs8w7C4PDREEBlvlbD24Mg==0ho="
    }, {
        "name": "wireless.wifinet0.macfilter",
        "value":
"ju3SOZneUDRpcLPbeCg0Vw==fxUBfORKNuR5Me8rLaKZEg==F7dW/9q
iAA=="
    }, {
        "name": "wireless.wifinet0.maclist",
        "value":
"xCHHCJGD1u3SIr2M1jLLJw==qiPay5WE9jG8sp0UiW8YsA=="
    }, {
        "name": "wireless.wifinet0.disabled",
        "value":
"L4LBCneE19hTQlTJ1MUasg==CdqFNqCvmwr12Vyiaf3XPw==bA=="
    }],
    "status": [{
        "name": "wifi.clients",
        "value": "Command failed: Not found"
    }, {
        "name": "amazert.heartbeat.time",
        "value": "Mon Nov 30 01:04:03 UTC 2020"
    }, {
        "name": "dhcp.leases",
        "value": {
            "dhcp_leases": [],
            "dhcp6_leases": []
        }
    }, {
        "name": "assoclist.wlan0",
        "value": {
            "results": []
        }
    }, {
        "name": "amazert.poweron.time",
        "value": "Mon Nov 30 01:04:03 UTC 2020"
    }, {
        "name": "amazert.status",
        "value": "running"
    }],
    "identifier": {
        "uid": "0NWFFJG764T2ucPAhdLDNwQjcgA2",
        "deviceId": "f0524c82-2614-11eb-b140-
dca6328f80c0",
        "email": "nabin.thomas@gmail.com"
    }
}
```