SAVONIA
University of Applied Sciences

THESIS – BACHELOR'S DEGREE PROGRAMME
TECHNOLOGY, COMMUNICATION AND TRANSPORT

AWS GREENGRASS IN BUILDING IOT APPLICATIONS

AUTHOR Tommi Lehikoinen

SAVONIA UNIVERSITY OF APPLIED SCIENCES THESIS
Abstract

Field of Study Technology, Communication and Transport	
Degree Programme Degree Programme in Internet of Things	
Author(s) Tommi Lehikoinen	
Title of Thesis AWS Greengrass in Building IoT Applications	
Date 25 May 2023	Pages/Appendices 42
Client Organisation Etteplan Oy	

Abstract

Edge computing is a paradigm that brings computation closer to the data source and can solve many challenges with IoT devices and their development. For applications with critical time constraints, computation at the edge provides faster response times compared to cloud computing. However, edge devices often need to maintain a connection to the cloud to access greater computational power or data storage. Edge devices also require scalability, remote upgradeability, and device state monitoring to be effective. This thesis aims to explore how AWS Greengrass can address these problems and be used to build IoT applications.

To demonstrate the capabilities of AWS Greengrass and its potential use in building IoT applications, a proof of concept (POC) was developed with it. After installing the Greengrass Core software on the Raspberry Pi computer, software was developed to provide control of the GPIO for input and output, remote monitoring and device state updates, and communication between different software modules. Connectivity was established to an ESP32 microcontroller in a local network, and the data from it was then forwarded to a cloud service for further processing and analytics.

In conclusion, AWS Greengrass and its features were successfully utilized to implement an IoT demo application, with the device, edge, and cloud layers each performing their suitable tasks. For future research, the Greengrass feature called Fleet Provisioning could be investigated for handling the horizontal scaling of the system. In this demo, the certificate keys to authorize devices were hardcoded, but using Fleet Provisioning would be a better approach for scalability.

Keywords
Amazon Web Services, Greengrass, edge computing, IoT development

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

The Internet of Things (IoT) has become increasingly prevalent in today's world, transforming the way devices and people interact with each other. IoT provides a new level of connectivity that is used to streamline and automate operations from smart homes to industrial factories by connecting electronic devices and sensors to each other through the internet. An IoT network is interconnected, allowing it to share data between different connected devices, sometimes referred to as "things". Data from one can be used to inform and optimize the operations of others in real-time and to create efficient systems. (Sachin, Tiwari, & Zymbler, 2019)

The level of connectivity the IoT creates has the potential to revolutionize our lives, but it also presents new challenges that must be addressed. According to Čolaković & Hadžialić (2018, 10-21), the continually increasing number of connected devices is at the forefront of these challenges. With the vast amount of data generated, limitations in computing power and network bandwidth become evident. Ensuring seamless availability of remotely located devices and managing their deployments and updates in a secure way are challenges that need to be overcome. Additionally, systems need to be scalable, and devices must be uniquely addressable. (Čolaković & Hadžialić, 2018)

To address some of these challenges, edge computing has emerged as a key solution. By processing and analyzing data at the edge of the network, closer to the source of the data, edge computing can reduce network latency, transmission costs, and improve security and privacy (Premsankar, Di Francesco, & Taleb, 2018). Companies like Amazon Web Services (AWS) offer services to help with creating infrastructure for IoT networks. AWS Greengrass, for example, is a service provided by AWS that extends AWS cloud capabilities to the edge of the network. Greengrass helps to address scalability issues by allowing users to manage and update devices remotely in groups and offering security features such as encryption and device authentication. (What is Greengrass?, 2023)

In this thesis, edge computing and its role in the broader IoT landscape is explored. AWS IoT Greengrass, its features, architecture, and deployment process are introduced. Also, development with the Greengrass is shown by creating a demo application with it. By the end of this thesis, a solid understanding of AWS IoT Greengrass and how it can be used to build and deploy edge computing solutions for IoT projects will have been gained.

This thesis was undertaken in partnership with Etteplan Oy, a Finnish engineering and technical documentation services company that offers a wide range of engineering, digitalization, and IoT services (Etteplan, 2023). The research conducted on AWS IoT Greengrass was aimed at improving the company's capabilities in providing commercial edge computing solutions.

2 EDGE COMPUTING

2.1 Edge computing in comparison to Cloud and Fog paradigms

Edge computing is a distributed computing paradigm that enables computation to be performed at the network edge, closer to the source of data. This approach stands in contrast to cloud computing which uses remote cloud data centers for storing and analyzing the data and performing computations. A third computing paradigm, known as fog computing, is a form of edge computing that stands between edge and cloud. Fog computing extends the idea of edge computation to a larger scale, by distributing processing and storage resources across the network, for edge devices to connect to. In a three-layer architecture (see Figure 1), devices at the edge connect to the fog layer, that acts as an intermediate layer between IoT devices and the cloud. (Angel, Ravindran, Vincent, Srinivasan, & Hu, 2021)

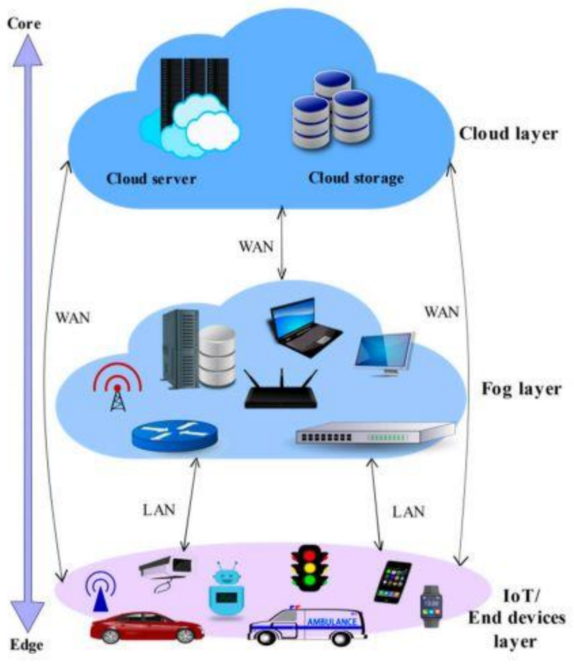


FIGURE 1. Three-layer architecture (Angel, Ravindran, Vincent, Srinivasan, & Hu, 2021)

Edge and fog computing share some similarities in the problems they address, such as reducing latency, improving response times, and minimizing the amount of data that needs to be sent to the cloud. However, edge computing performs computations, such as filtering and analyzing, directly on the measuring device itself, which can range from a simple sensor to a more complex edge server. In contrast, fog computing acts as a computational resource to which multiple edge devices can connect. Fog computing is often used in situations where many edge devices need to be connected and managed, and where there is a need for more scalable and flexible computing resources that can adapt to changing workloads and network conditions. (Angel, Ravindran, Vincent, Srinivasan, & Hu, 2021)

2.2 Evolution of edge computing

The concept of bringing computational power closer to the source has been around for some time. A company called Akamai pioneered edge computing by developing Content Delivery Networks (CDNs) in the late 1990s to distribute network content more efficiently. CDNs accomplished this by using proxy servers with caching capabilities, which allowed them to provide faster response times to users by serving content from nodes located closer to the end-users. This approach not only improved the user experience but also helped reduce network congestion and bandwidth costs. (Iftikhar, et al., 2023)

The concept of moving computation and data storage closer to the end-user has evolved into what we now know as edge computing, and is critical for a wide range of applications, including remote healthcare monitoring and industrial control systems that require low-latency communication. (Premsankar, Di Francesco, & Taleb, 2018).

Nowadays, edge computing allows IoT devices to utilize Artificial Intelligence (AI) and Machine Learning (ML) algorithms in real-time for the improvement of many aspects of human life. For example, deep-learning frameworks for creating medical images have been developed and can be used together with data from wearable healthcare devices to find important information about a person's health conditions. (Iftikhar, et al., 2023)

2.3 Role of edge computing in IoT

Internet of Things (IoT) devices create a digital representation of our physical world in machine-friendly data. However, the expected amount of data generated by these devices is not designed to be handled by traditional cloud infrastructure and can lead to performance issues, especially with time-sensitive applications. Edge computing is a necessary paradigm for creating efficient IoT architecture, as it solves many of these problems. By filtering data at the edge, several benefits can be achieved, including reduced network traffic, lower latency for communication, and reduced cloud costs by using less storage space. (Iftikhar, et al., 2023)

In addition, a study by Sharifi, Rameshan, Freitag, & Veiga (2014) compared energy consumption between decentralized P2P-cloud and datacenter solutions and found that the closer the source of data and server are, the less energy is consumed in the network. This is an important consideration for IoT devices, which are usually constrained by their energy capabilities. By utilizing the edge instead of traditional cloud systems, energy consumption can be minimized, providing an additional advantage for edge computing in IoT applications.

3 AWS IOT AND GREENGRASS

3.1 Introduction to AWS IoT

Amazon Web Services (AWS) offers a wide variety of software services specifically targeted at the Internet of Things (IoT). AWS categorizes these into three distinct sections. Device software runs at the edge and is used to build the sensing layer of the IoT network. Connectivity and control services connect edge devices to the cloud and have tools for organizing and managing them. Analytics services are used to extract value from data and have tools for responding to certain events and keeping track of real-world devices by creating a digital twin, for example. (AWS IoT, 2023)

Combining these services with other parts of the AWS cloud can be used to build, for example, an event-driven architecture as shown in Figure 2. In this architecture example, IoT sensor data is collected using AWS IoT Greengrass which responds to local events from sensors and communicates with the cloud securely. AWS IoT Core, which handles communication with the edge, further sends this information to Amazon Kinesis which analyzes data in near real-time. Results from analysis are then stored in cloud storage and an AWS Lambda function is triggered that invokes an Amazon SageMaker model for machine learning. (Sodabathina, Shan, & Ulloa, 2022)

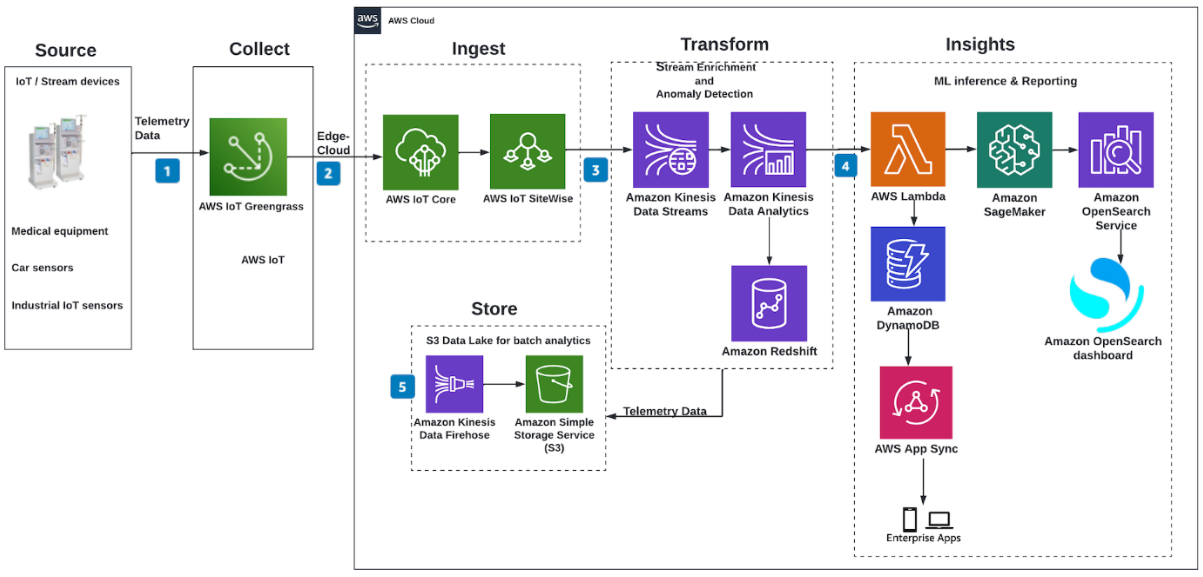


FIGURE 2. AWS IoT event-driven architecture (Sodabathina, Shan, & Ulloa, 2022)

Services provided by AWS can be used in the creation of smart end-to-end IoT solutions that connect billions of devices together and extract value from the gathered data. Its different edge and cloud services are intended to be integrated with each other, and the tools AWS provides for scalability and security make it a good platform for implementing an IoT project. In the next chapters the focus is on AWS IoT Greengrass, which is only a small part of the IoT pipeline. Understanding the capabilities of the AWS cloud is important as Greengrass extends some of these features to edge devices.

3.2 Greengrass definition

AWS IoT Greengrass (Greengrass) is an open-source software service provided by Amazon Web Services (AWS). Greengrass consists of cloud service and client software as shown in Figure 3. Cloud services allow developers to build, deploy and manage software on their devices. These devices run Greengrass client software and can be managed in scalable groups. Client software is a combination of edge runtime and cloud service as it extends AWS cloud capabilities to remote environments without the need for a constant internet connection. This means that AWS cloud functionalities such as performing machine learning analytics or running Lambda functions can be executed directly on the device. Client software can be used to provide real-time responses to the devices that are connected to it and to control what data is sent to the cloud for further analytics and storage. In a situation where the Internet connection fails client devices continue to perform and can be set to synchronize when the connection re-appears. (AWS Greengrass, 2023)

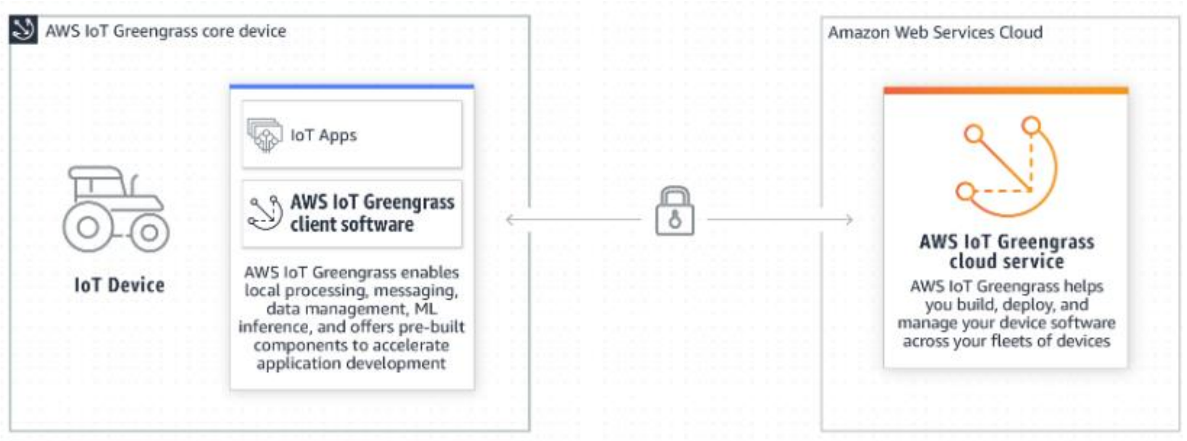

AWS IoT Greengrass core device
IoT Device
IoT Apps
AWS IoT Greengrass client software
AWS IoT Greengrass enables local processing, messaging, data management, ML inference, and offers pre-built components to accelerate application development
Amazon Web Services Cloud
AWS IoT Greengrass cloud service
AWS IoT Greengrass helps you build, deploy, and manage your device software across your fleets of devices

FIGURE 3. AWS Greengrass consists of cloud service and client software (AWS Greengrass, 2023)

Greengrass capitalizes on edge computing. It is an example of how edge computing can reduce unnecessary cloud traffic for faster responses and how cloud can be used on the back, for device management and updates, processing power and backup storage when needed.

3.3 Greengrass features

3.3.1 Local processing

AWS IoT Greengrass (Greengrass) acts as a local cloud service on edge premises. By bringing the AWS IoT Device SDK to the edge, other local devices can securely connect to each other to share messages or their state over the communication path (AWS IoT SDK, 2023). Other typical AWS Cloud functionalities, such as executing serverless Lambda functions to respond to events or managing and running Docker containers, are also supported by Greengrass. Access to device resources, such as General-Purpose Input/Output (GPIO), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C) and other serial ports are also supported. (AWS IoT Greengrass Features, 2023)

3.3.2 Machine learning inference

AWS Greengrass supports the execution of pre-trained machine learning models to create predictions on new data, often referred to as inferencing. Statistical algorithms used in the creation of these models require significant computing resources making the cloud a well-suited option for them. However, as inferencing does not require as much computing power, performing predictions on a lower-powered device is possible. AWS Greengrass can be used to transfer trained models from the cloud to an edge device and to perform predictions locally with fast response times. Edge devices can also send gathered data from inference back to cloud to improve the quality of machine learning model. (ML Inference, 2023)

3.3.3 Data stream management

AWS IoT Greengrass has a stream manager component that provides an interface for transferring data to the AWS cloud, where it can be stored in services such as S3 or used by services like AWS IoT analytics. Stream manager is a reliable way of transferring high volumes of data to cloud. Users can define how network disruptions are handled to possibly resend data to cloud after connection restores. Also, policies can be defined that specify how handling of different types of data, such as prioritizing critical data to be sent first or limiting the bandwidth usage for certain types of data. (Data Streams, 2023)

3.3.4 Device shadows

Device shadows are virtual representations of a device’s state, encoded in JSON format. Shadows can be used to control, change, and access a device’s state. Greengrass core devices can connect to the AWS IoT device shadows service via its components. There are three methods for controlling device shadows: creation, updating, and deletion. These actions can be performed either from the AWS IoT Console, web clients, or other connected devices. (Device Shadows, 2023)

3.3.5 Managing groups of devices

Greengrass core devices and client devices can be combined in Greengrass Groups. For instance, a factory may group devices on different floors into separate groups. Alternatively, devices may be grouped based on their function, such as their responsibility at the assembly line. The power of grouping is in their management. Over the air updates can be targeted at certain groups, allowing all devices to be updated when needed. (AWS IoT Greengrass Features, 2023)

3.4 Greengrass architecture

3.4.1 Core and client devices

In an AWS IoT Greengrass architecture, a core device refers to a device that runs the AWS IoT Greengrass Core software. The core device serves as the hub of the IoT network at the edge, managing communication with the cloud and client devices. An example of a client device is a microcontroller running FreeRTOS that communicates with the core device using the MQTT protocol. The core device can act on the data it receives from the client device and, for example relay the data to another client device or the AWS IoT Core cloud service. (How AWS IoT Greengrass Works, 2023)

3.4.2 Greengrass component

Greengrass components are software modules that run on a core device. Components are composed of a recipe and artifacts. A recipe is written in JSON or YAML format and it defines component's metadata, configuration parameters, dependencies, and its lifecycle from installation to run and shutdown. Files such as binaries, scripts, and other files the component consumes are called artifacts. When a component needs to use functionality provided by another component, it becomes a dependent component. For instance, a runtime installer can act as a dependency for an executable file. (Greengrass components, 2023)

AWS IoT Greengrass provides readymade components anyone can use and deploy to their devices. Some of these components are open source which allows users to customize the source code (Public components, 2023). Greengrass community has also developed components which are free to use and open source (Community components, 2023). A GitHub repository (Greengrass Software Catalog, 2023) is an index for community components. Using readymade components can speed up the process of development. As an example, the community repository has a component called "S3 file uploader", which transfers new files in a core device directory to a S3 storage using a Greengrass Stream manager component. Another example for an AWS provided component is a "AWS Greengrass Shadow Manager", that can be used to transfer a virtual representation of devices state, a shadow, between different components locally or the AWS IoT Device Shadow service in the cloud. Components can also be created from scratch and some examples of recipes and artifacts is found in chapter 5.3 and its subchapters where the core device functionality is discussed.

3.5 Hardware requirements

Greengrass officially supports Linux environments with the following architectures: Armv7l, Armv8, and x86_64. Greengrass can also be installed in Windows environments with x86_64 architecture, with the caveat that some features are not supported. This thesis focuses on Linux based systems.

AWS IoT Greengrass (Greengrass) documentation for setting up core devices (Setting up core devices, 2023) lists supported platforms and other device requirements required for running Greengrass Core software.

To determine if a Linux-based IoT device and its CPU architecture, Linux kernel configuration, and installed drivers support Greengrass and interoperate with AWS IoT services, the AWS IoT Device Tester (Device tester, 2023) can be used to perform device validation.

To extend Greengrass to Linux devices regardless of the hardware architecture, it is possible to build a custom Linux-based operating system with a tool such as Yocto and integrate BitBake recipes from meta-aws (meta-aws, 2023) into it.

4 AWS GREENGRASS DEVELOPMENT AND DEPLOYMENT

4.1 Raspberry Pi 4

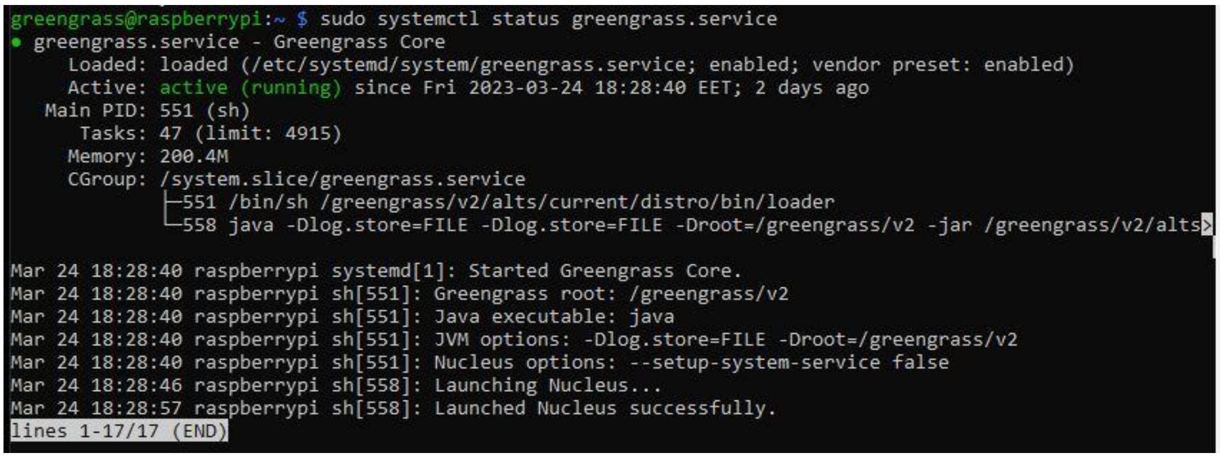
For this thesis, a demo project was done to explore the features of AWS Greengrass in building an IoT application. Raspberry Pi 4 Model B (Raspberry) is a single-board computer that is built on the Armv8 architecture. The board is equipped with an Ethernet port, Wi-Fi, and Bluetooth support for connectivity, and it has a 40-pin GPIO header that can be used to connect various sensors and electronic components to it (Raspberry Specifications, 2023). Due to its features, it is a viable option when creating proof of concepts for projects, and since it meets the necessary requirements to function as an AWS Greengrass core device, it was selected as the development board for this thesis. The Raspberry Pi 32-bit Operating System (OS) was installed on it using the Raspberry Pi Imager tool.

4.2 Requirements

Greengrass core devices require AWS IoT and Identity and Access Management (IAM) provisioning to connect the core device to the cloud. IAM is a service for managing identities and their access to AWS services and resources by granting users and roles permissions to specific resources (AWS IAM, 2023). Every AWS IoT Greengrass device needs to be given policies to perform tasks such as connecting to AWS IoT Core or to enable client devices to use Greengrass cloud discovery to find their core device (Installing Greengrass, 2023). In addition to policies, Greengrass devices also need a digital certificate for authentication, and AWS uses X.509 certificates to ensure a secure connection between devices (Device authentication, 2023).

There are four different ways to install the necessary policies and authentication certificates on the core device: Quick installation requires the fewest steps and uses an installer that creates the necessary resources with the AWS credentials provided to it. Manual installation requires resources to be manually created but it can be used to install devices that are behind a firewall or a proxy, and certificates can optionally be saved into cryptographic elements such as a hardware security model (HSM). AWS IoT Fleet provisioning allows multiple devices to be set up in the same way and suits large-scale deployments. Custom provisioning gives more control over creating necessary resources, and can be customized, for example, to provide custom X.509 certificates. Additionally, programs such as the Java runtime need to be installed and some configurations need to be changed. AWS provides all the requirements and steps for installing them in their documentation. (Installing Greengrass, 2023).

Quick installation was used to configure the Raspberry as a core device. After completing AWS instructions for setting up a core device (Automatic provisioning, 2023), Core software was successfully running as shown in Figure 4. During the installation a name for the device "ThesisCoreDevice1" was given and it was also attached to the thing group "ThesisCoreDevices".



```
greengrass@raspberrypi:~ $ sudo systemctl status greengrass.service
● greengrass.service - Greengrass Core
     Loaded: loaded (/etc/systemd/system/greengrass.service; enabled; vendor preset: enabled)
     Active: active (running) since Fri 2023-03-24 18:28:40 EET; 2 days ago
   Main PID: 551 (sh)
      Tasks: 47 (limit: 4915)
     Memory: 200.4M
     CGroup: /system.slice/greengrass.service
             ├─551 /bin/sh /greengrass/v2/alts/current/distro/bin/loader
             └─558 java -Dlog.store=FILE -Dlog.store=FILE -Droot=/greengrass/v2 -jar /greengrass/v2/alts>

Mar 24 18:28:40 raspberrypi systemd[1]: Started Greengrass Core.
Mar 24 18:28:40 raspberrypi sh[551]: Greengrass root: /greengrass/v2
Mar 24 18:28:40 raspberrypi sh[551]: Java executable: java
Mar 24 18:28:40 raspberrypi sh[551]: JVM options: -Dlog.store=FILE -Droot=/greengrass/v2
Mar 24 18:28:40 raspberrypi sh[551]: Nucleus options: --setup-system-service false
Mar 24 18:28:46 raspberrypi sh[558]: Launching Nucleus...
Mar 24 18:28:57 raspberrypi sh[558]: Launched Nucleus successfully.
lines 1-17/17 (END)
```

FIGURE 4. Core software running on Raspberry Pi 4

4.3 Development tools

Greengrass Core components can be developed either on the core device or in a local development computer (Create components, 2023). When components are developed on the core device, they can be tested right away without the need to use the Greengrass cloud service to first publish them. AWS offers several tools to support Greengrass device development, including the Greengrass Development Kit Command-Line Interface (GDK CLI), Greengrass Command Line Interface (Greengrass CLI), and Local debug console (Development tools, 2023).

The GDK CLI provides developers with a command-line interface for creating components from templates or community components. The tool can be used to create and build components and publish them to the AWS IoT Greengrass service. GDK CLI automatically updates the component's version and artifact URIs to the recipe file when a new version is created. (GDK CLI, 2023)

The Greengrass CLI is a command-line interface tool that runs on the core device, allowing developers to develop components locally and deploy and debug them without requiring the use of the Greengrass cloud service (Greengrass CLI, 2023). The tool can be deployed to the core device as a component, as shown in Chapter 4.5.

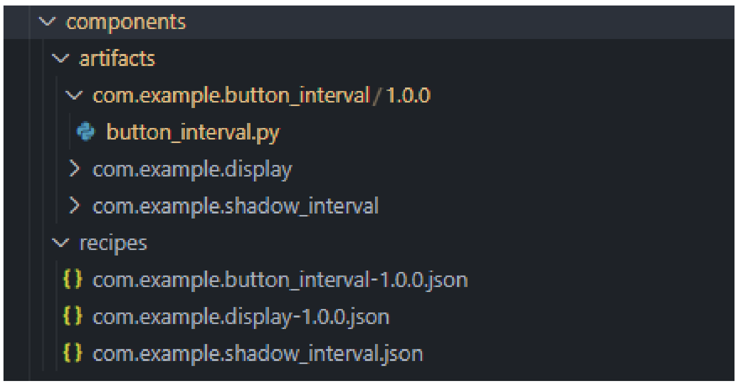
The local debug console component is a local dashboard for viewing and managing components in a core device. It shows components and their dependencies and has tools for changing the running config of a component to speed up the development process. (Local debug console, 2023)

Another essential tool for any AWS development is the AWS Command Line Interface (AWS CLI). It is a tool for accessing and managing AWS services.

4.4 Development

4.4.1 Developing components locally

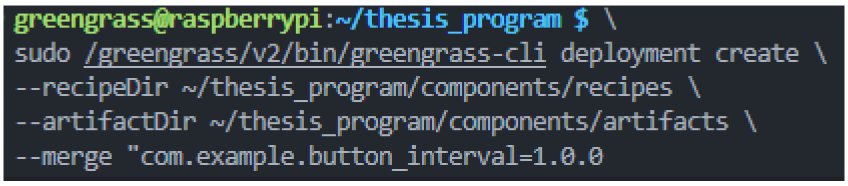
To create components locally, one option is to have a folder for artifacts and another one for recipes, as demonstrated in Figure 5. Components use semantic versioning that follows a major.minor.patch number system, and each version has its own recipe and artifact folder. Figure 5 shows how three components each have their own artifact folders with a specific folder for version, and a recipe that also specifies the version it belongs to.



```
components
  artifacts
    com.example.button_interval / 1.0.0
      button_interval.py
    com.example.display
    com.example.shadow_interval
  recipes
    com.example.button_interval-1.0.0.json
    com.example.display-1.0.0.json
    com.example.shadow_interval.json
```

FIGURE 5. Example of a component folder structure

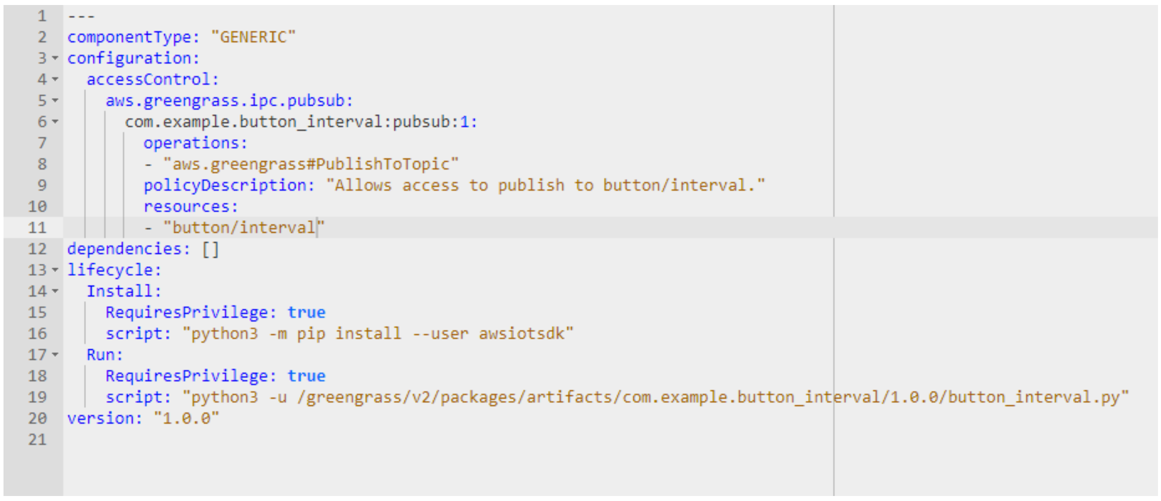
Once the development of the component is complete, it can be deployed to the device locally using the Greengrass CLI’s command for deployment, as shown in Figure 6. In this command a folder for recipes and artifacts, along with the component to be merged is given as an argument to the deployment command.



```
greengrass@raspberrypi:~/thesis_program $ \
sudo /greengrass/v2/bin/greengrass-cli deployment create \
--recipeDir ~/thesis_program/components/recipes \
--artifactDir ~/thesis_program/components/artifacts \
--merge "com.example.button_interval=1.0.0
```

FIGURE 6. Deploying component locally using Greengrass CLI software

After successful deployment, the component is run by Greengrass core software and it also appears in the local debug console, where its running configuration can be modified to make fast changes to its configuration. Another option to make changes to the configuration of the deployed component is to remove it from the Greengrass Core software and then redeploy it with the desired configuration. However, this process takes a longer time compared to modifying the running configuration in the local debug console. Figure 7 shows an example of a running configuration in the local debug console.



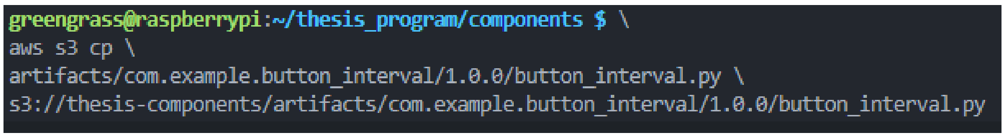
```yaml
---
componentType: "GENERIC"
configuration:
  accessControl:
    aws.greengrass.ipc.pubsub:
      com.example.button_interval:pubsub:1:
        operations:
        - "aws.greengrass#PublishToTopic"
        policyDescription: "Allows access to publish to button/interval."
        resources:
        - "button/interval"
dependencies: []
lifecycle:
  Install:
    RequiresPrivilege: true
    script: "python3 -m pip install --user awsiotsdk"
  Run:
    RequiresPrivilege: true
    script: "python3 -u /greengrass/v2/packages/artifacts/com.example.button_interval/1.0.0/button_interval.py"
version: "1.0.0"
```

FIGURE 7. Local debug console and a running config of a component

Another option to develop components locally is to use the GDK CLI tool and its initialization command. Using, for example, a "HelloWorld" template gives a good starting point for component development. GDK CLI's build and publish commands upload components and artifacts to the Greengrass Cloud service automatically. The GDK CLI, however, does not transfer recipe's access control policies to the cloud, so they must be reconfigured each time component is deployed to a device. An example of a component that was developed with the GDK CLI is shown in chapter 5.3.3.

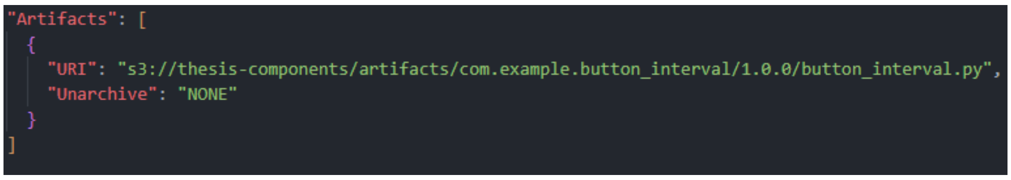
4.4.2 Publishing a component

To be able to deploy a component to other core devices over the internet, it must first be published by uploading it to the Greengrass Cloud Service. If the component is created using the GDK CLI tool, it can be published using the tool's publish command. Another option is to upload artifacts and recipes to the cloud manually using AWS shell commands. As the Raspberry Pi has a 32-bit OS, AWS CLI V1 was installed as the supported command-line interface. Then the artifact file was copied to the S3 (Figure 8) using the AWS CLI command. The recipe was modified by adding information about the artifact location in the S3 (Figure 9), and then the component was created and published from the recipe file using another AWS CLI command (Figure 10).



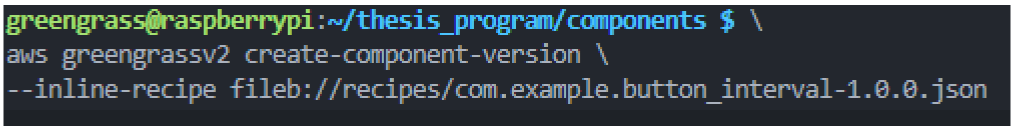
```
greengrass@raspberrypi:~/thesis_program/components $ \
aws s3 cp \
artifacts/com.example.button_interval/1.0.0/button_interval.py \
s3://thesis-components/artifacts/com.example.button_interval/1.0.0/button_interval.py
```

FIGURE 8. Copying artifact to S3 cloud storage



```
"Artifacts": [
  {
    "URI": "s3://thesis-components/artifacts/com.example.button_interval/1.0.0/button_interval.py",
    "Unarchive": "NONE"
  }
]
```

FIGURE 9. Artifact's path added to the recipe



```
greengrass@raspberrypi:~/thesis_program/components $ \
aws greengrassv2 create-component-version \
--inline-recipe fileb://recipes/com.example.button_interval-1.0.0.json
```

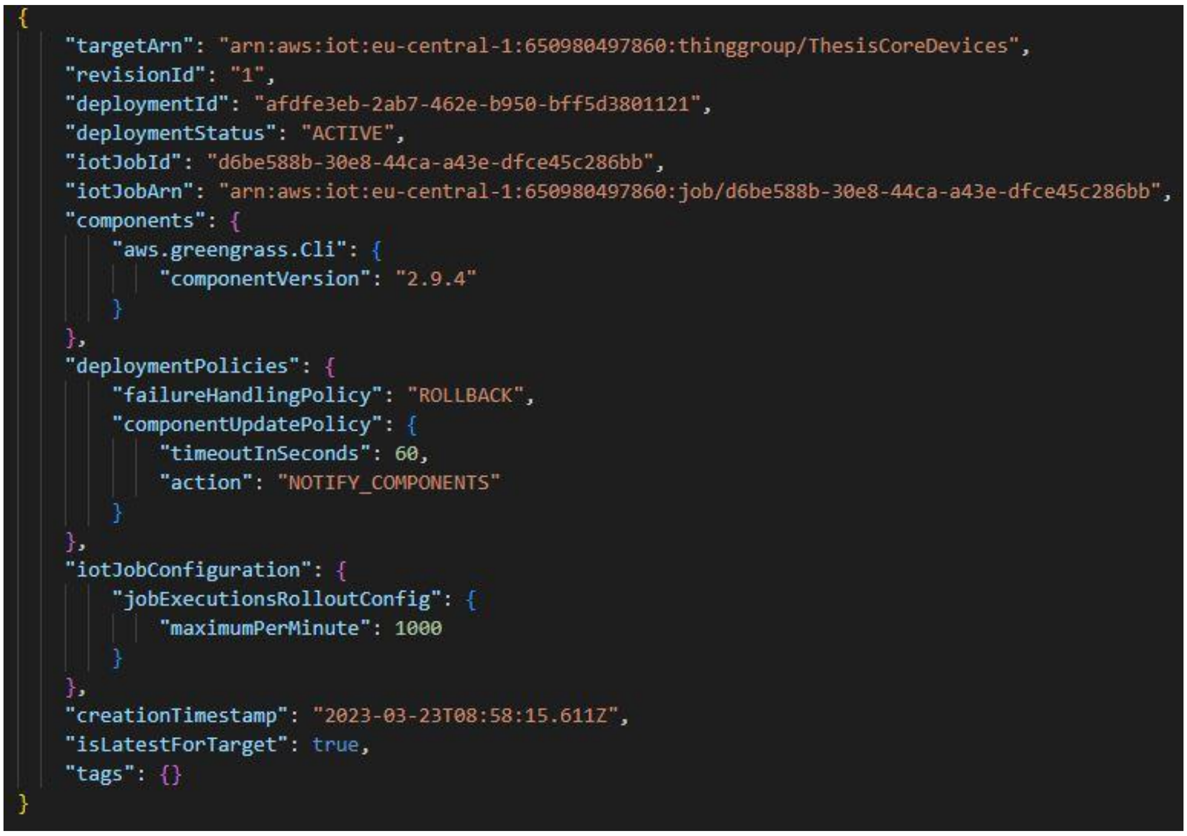
FIGURE 10. Creation of the component

4.5 Deployment

Components that are accessible from the Greengrass cloud service can be deployed to Greengrass core devices by creating a deployment, which involves choosing the components and their configurations with the target that defines the devices or groups of devices to receive the deployment. Deployments use IoT Jobs, so the strategy for the rollout of deployments can be managed. Deployments are continuous, which means that devices that are not connected and cannot receive the deployment right away will receive it the next time they connect to the Greengrass cloud service. The target can only have one deployment at a time, and previous deployments are overwritten on new deployments, however the AWS Greengrass service stores previous versions of the deployments and revising or rolling back to certain versions is possible. (Deployment, 2023)

There are several deployment configuration options available in Greengrass. Rolling updates involve updating a subset of devices at a time with either a constant rate or exponential rate. Custom deployment strategies allow for more complex deployment scenarios, such as stopping the deployment rollout if a certain percentage of a certain number of deployments fail to succeed (Jobs, 2023). When the deployment is received by a core device, each component is notified by default. Components can be customized to respond to these events based on their state for example, a component tracking the battery level can defer the update if battery level is below certain threshold (Deployment, 2023). During the deployment process, the Greengrass platform automatically manages the component's lifecycle, handling tasks such as starting and stopping the component, managing dependencies, and handling errors. Error policies can be set to, for example, roll back to previous version if one component fails to start. This ensures that components are running smoothly on the core device and that updates are delivered efficiently and reliably.

Greengrass CLI was deployed on the Raspberry to ease the development of the components. Using the AWS IoT Console, the component was selected from the list of public components (Public components, 2023) and a deployment was created to target the thing group Raspberry Pi was attached to. If multiple core devices had been in the group all the devices would have got the deployment. The deployment configuration is shown in Figure 5 below. The deployment was created using the AWS console, which has a user interface for creating the deployment configuration.



```json
{
    "targetArn": "arn:aws:iot:eu-central-1:650980497860:thinggroup/ThesisCoreDevices",
    "revisionId": "1",
    "deploymentId": "afdfe3eb-2ab7-462e-b950-bff5d3801121",
    "deploymentStatus": "ACTIVE",
    "iotJobId": "d6be588b-30e8-44ca-a43e-dfce45c286bb",
    "iotJobArn": "arn:aws:iot:eu-central-1:650980497860:job/d6be588b-30e8-44ca-a43e-dfce45c286bb",
    "components": {
        "aws.greengrass.Cli": {
            "componentVersion": "2.9.4"
        }
    },
    "deploymentPolicies": {
        "failureHandlingPolicy": "ROLLBACK",
        "componentUpdatePolicy": {
            "timeoutInSeconds": 60,
            "action": "NOTIFY_COMPONENTS"
        }
    },
    "iotJobConfiguration": {
        "jobExecutionsRolloutConfig": {
            "maximumPerMinute": 1000
        }
    },
    "creationTimestamp": "2023-03-23T08:58:15.611Z",
    "isLatestForTarget": true,
    "tags": {}
}
```

FIGURE 11. JSON configuration for AWS CLI deployment

Referring to the deployment configuration shown in Figure 11, the following details can be observed:

- Deployment targets a thing group with a name "ThesisCoreDevices"
- Deployment deploys one component with a name "aws.greengrass.Cli"
- The component update timeout is 60 seconds, if update takes longer then update fails
- The rollout configuration is set to a maximum of 1000 devices per minute
- Deployment rolls back to the previous version on failure

5 IMPLEMENTING AN AWS IOT APPLICATION

5.1 Overview

The goal of the project is to use Greengrass and its features to demonstrate how they can be used in creating an IoT application that contains a sensor and actuator layer, an edge device, and a cloud service, with emphasis on the following topics:

- Scalability and over-the-air updates
- Monitoring and controlling device state over the internet
- Synchronization of data and state after network disruptions
- Edge computing and local responses to the events
- Data transfer to the cloud and forwarding it to other AWS services

As the project is centered around the utilization of AWS Greengrass software, with sensors and controllable devices playing a supporting role. The devices were chosen with simplicity in mind, but their specific features and capabilities are not the primary focus of the project. Instead, the focus is on showcasing the power and flexibility of the Greengrass platform. The proof of concept using Greengrass was built to provide the following functionalities:

- A Raspberry Pi as an edge device with Greengrass Core software installed
- An ESP32 microcontroller that connects locally to the core device and sends data to it
- A screen attached to the Raspberry Pi to display data from ESP32 and a time in intervals
- Saving the data to a database in the AWS cloud
- Control of the screen interval and ESP32 measurement cycle with a device shadow

Figure 12 below shows the architecture of the project and the devices with their connections.

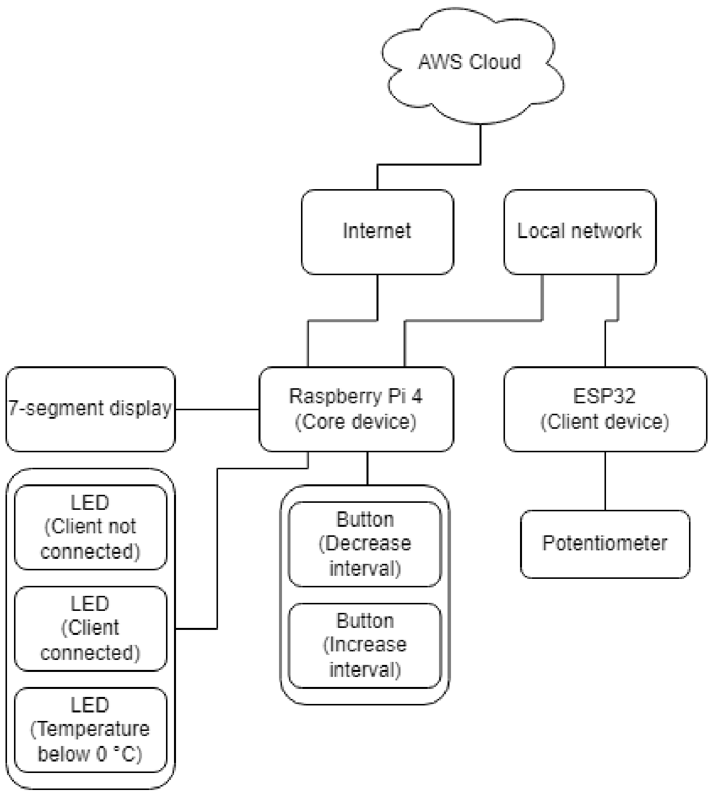

AWS Cloud
Internet
Local network
7-segment display
Raspberry Pi 4 (Core device)
ESP32 (Client device)
LED (Client not connected)
LED (Client connected)
LED (Temperature below 0 °C)
Button (Decrease interval)
Button (Increase interval)
Potentiometer

FIGURE 12. IoT application architecture

5.2 ESP32

5.2.1 Overview

The ESP32 is a powerful microcontroller with integrated Wi-Fi (ESP32, 2023) making it a suitable client device to be connected to the Greengrass core device. With support for MicroPython (MicroPython, 2023) it was chosen as a runtime for the client program. Higher-level languages such as Python can have faster development processes making it a practical choice when building proof-of-concepts.

The main objective of the microcontroller was to measure sensor data and transmit it to the core device. To achieve this with edge computing in mind, the connection was established within a local network. The benefits of this approach are that even if the global internet is unavailable, data transmission can still occur, and the core device can respond to these events with a fast response time. Additionally, control of the client device over the cloud was added to make the system more customizable. Figure 13 below displays the flowchart of the ESP32 program.

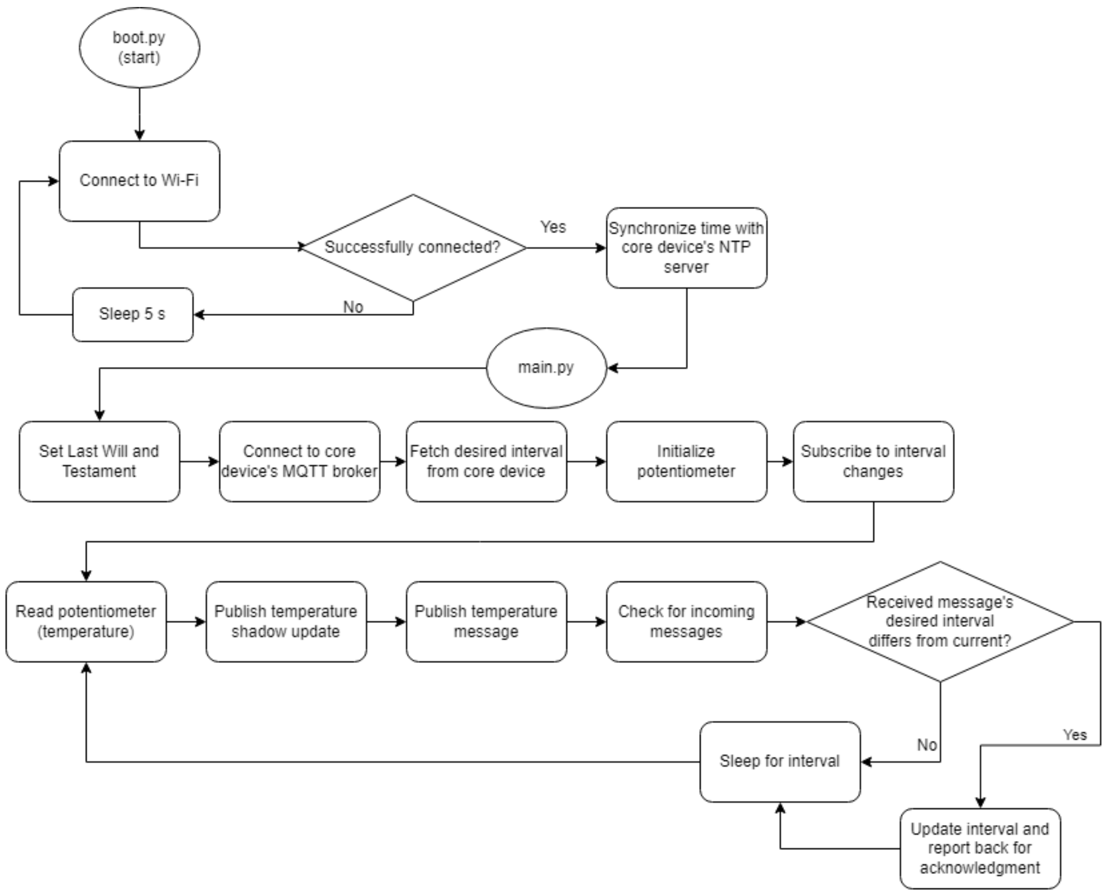
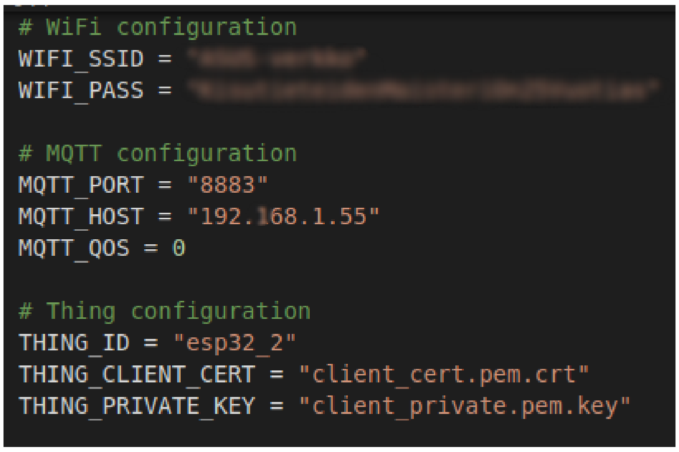

boot.py (start)
Connect to Wi-Fi
Successfully connected?
Yes
No
Sleep 5 s
Synchronize time with core device's NTP server
main.py
Set Last Will and Testament
Connect to core device's MQTT broker
Fetch desired interval from core device
Initialize potentiometer
Subscribe to interval changes
Read potentiometer (temperature)
Publish temperature shadow update
Publish temperature message
Check for incoming messages
Received message's desired interval differs from current?
No
Yes
Sleep for interval
Update interval and report back for acknowledgment

FIGURE 13. ESP32 program flowchart

5.2.2 Initial setup

MicroPython v.1.19.1 ESP32 firmware (ESP32 MicroPython, 2023) was installed by following instructions from the MicroPython documentation (ESP32 tutorial, 2023). MicroPython firmware is a combiled .bin image and it was downloaded to the ESP32 using a program named esptool (Esptool, 2023).

The ESP32 was registered as a thing on the AWS IoT Core platform and the thing was added as a client device for the core device. The registration process generates a certificate and a private key for the device. The certificate and private key were uploaded to the ESP32 to be used for authenticating the client device with the core device. The Wi-Fi Router's SSID and password were hardcoded into the code, along with information about the core device and its IP address and MQTT port as shown in Figure 14 below.

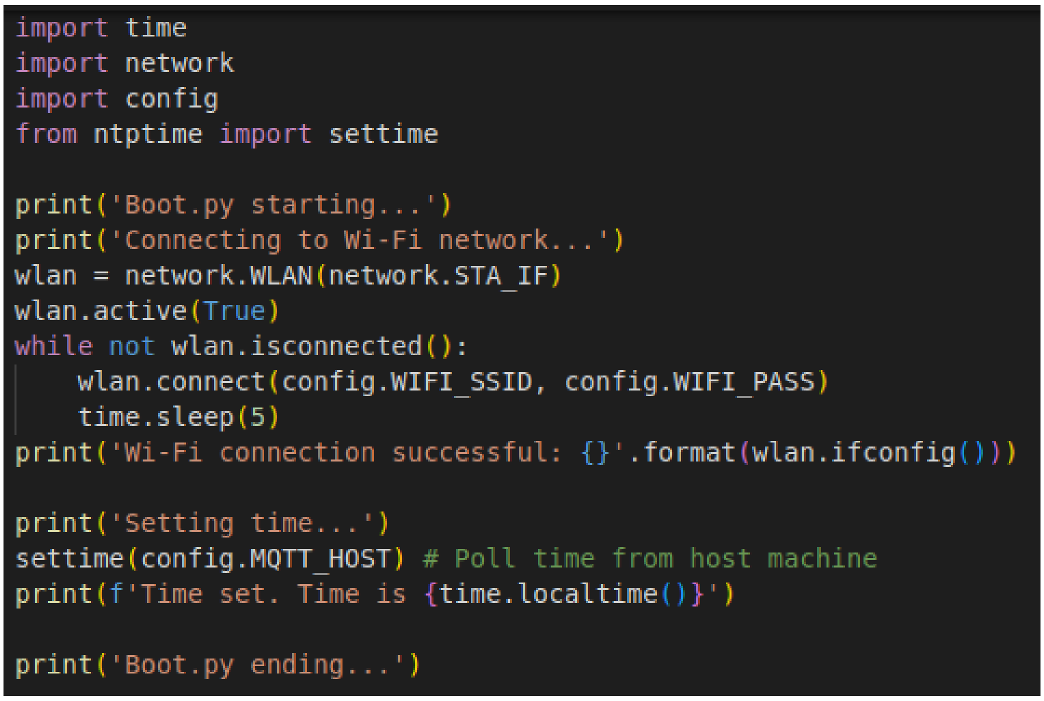


```
# WiFi configuration
WIFI_SSID = [redacted]
WIFI_PASS = [redacted]

# MQTT configuration
MQTT_PORT = "8883"
MQTT_HOST = "192.168.1.55"
MQTT_QOS = 0

# Thing configuration
THING_ID = "esp32_2"
THING_CLIENT_CERT = "client_cert.pem.crt"
THING_PRIVATE_KEY = "client_private.pem.key"
```

FIGURE 14. ESP32 config file

5.2.3 Connecting to Wi-Fi and updating time

When MicroPython runs it executes two scripts, "boot.py" and "main.py" in respective order. The "boot.py" is typically used to set up hardware and system-level configurations that need to be done before running the main application. As Figure 15 shows below, A Wi-Fi connection was established in this file using the credentials from the config file with the network-library. The program tries to initialize connection every 5-second interval until a successful connection is made. After the connection succeeds, the program proceeds to poll the current time from a Network Time Protocol (NTP) server that is running on the core device. By updating the time with the core device's NTP server, the client device can synchronize its clock without requiring access to the internet. By synchronizing the clock with the core device's NTP server, messages sent from the client device can include a timestamp that accurately reflects the time when the message and sensor data were generated. As the time is only requested once at startup, the program does not handle the possible issues that clock drift causes. Clock drift is a phenomenon where two clocks do not run exactly at the same rate, that causes the clocks to go out of sync. This could be solved by refreshing the time at regular intervals, for example, the program could be modified to request the time from the NTP server every 15 minutes.



```
import time
import network
import config
from ntptime import settime

print('Boot.py starting...')
print('Connecting to Wi-Fi network...')
wlan = network.WLAN(network.STA_IF)
wlan.active(True)
while not wlan.isconnected():
    wlan.connect(config.WIFI_SSID, config.WIFI_PASS)
    time.sleep(5)
print('Wi-Fi connection successful: {}'.format(wlan.ifconfig()))

print('Setting time...')
settime(config.MQTT_HOST) # Poll time from host machine
print(f'Time set. Time is {time.localtime()}')

print('Boot.py ending...')
```

FIGURE 15. Boot.py connects to Wi-Fi and updates system time

5.2.4 MQTT

MQTT library for MicroPython (umqtt.simple, 2023) was used to establish a connection between the client and the core device. Generated certificate file and key were used as SSL parameters to ensure secure communication. MQTT's Last Will and Testament (LWT) feature is a mechanism that allows a client device to specify a message that will be published to a specific topic on the MQTT broker if the client disconnects from the broker unexpectedly. As LWT must be specified when the connection is established, an LWT message that reports offline status was added to the instance prior to connecting with the MQTT broker. This way, the core device's MQTT broker could store the received LWT message after the client connected and hold its release until communication with the client failed.

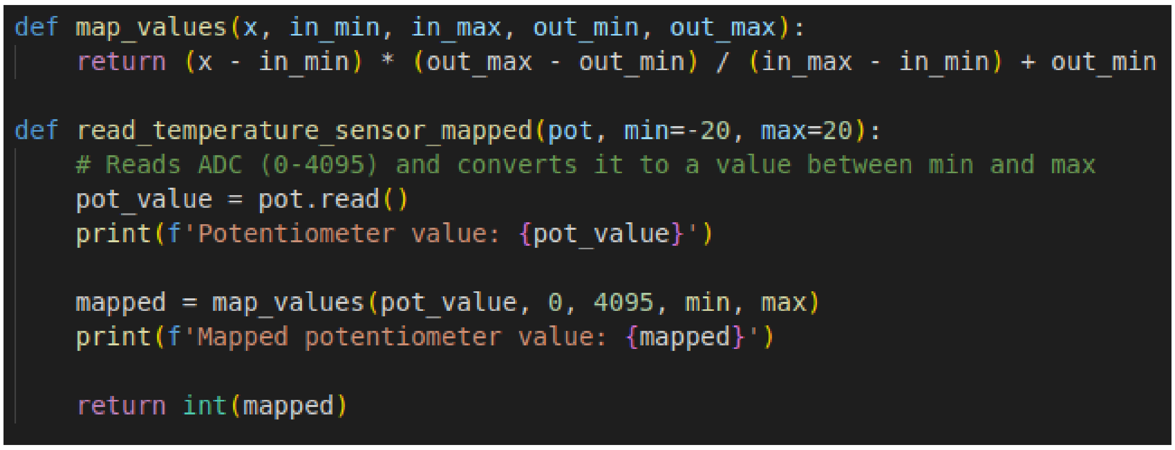
MQTT was used to communicate on the following topics:

- Publishing shadow updates for temperature and device state
- Publishing and subscribing to a shadow change for the sleep interval
- Publishing a message for temperature

5.2.5 Main program

The main program connects to the core device's MQTT broker to publish and receive messages on the desired topics. A measurement of the sensor is done at intervals. When the program first runs, it asks the core device for the current desired state of the interval. This is important to ensure the client device has a correct state after startup. An interval variable controls the time the program sleeps between the measurements. If a message to update the interval is received by a subscribed topic, the device updates its interval cycle after concurring sleep has been slept. Then the system reports back the changed interval to notify that the update was successfully received.

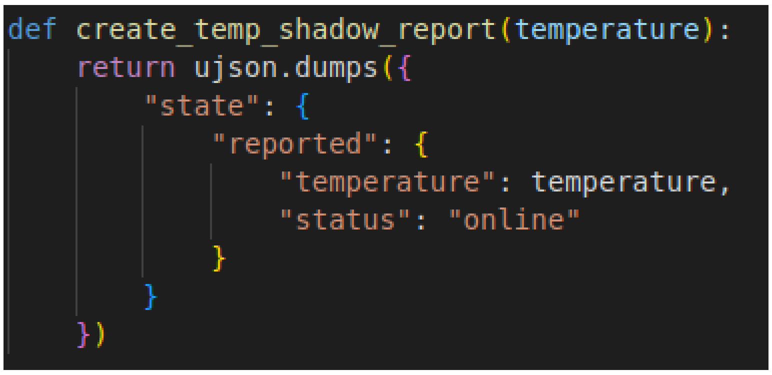
To have greater control over the development process and test the system's response to different sensor output values, a potentiometer was connected to the ESP32 board. The potentiometer provides a variable resistance that can be adjusted to simulate a wide range of sensor values. The ESP32's analog input pins offer a resolution of 12 bits, which provides a range of values between 0 and 4095 when the input is read. To better simulate a temperature sensor, a custom function was created to map the potentiometer's output to a specific integer range (see Figure 16).



```python
def map_values(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min

def read_temperature_sensor_mapped(pot, min=-20, max=20):
    # Reads ADC (0-4095) and converts it to a value between min and max
    pot_value = pot.read()
    print(f'Potentiometer value: {pot_value}')

    mapped = map_values(pot_value, 0, 4095, min, max)
    print(f'Mapped potentiometer value: {mapped}')

    return int(mapped)
```

FIGURE 16. Function to read potentiometer and convert value to a given range

The messages published on MQTT topics are sent as a JSON string. As an example, Figure 16 shows a function that returns a JSON string message that is used to report the current shadow state for temperature and status. In a similar way, the LWT message that reports offline status uses the same kind of message, but only with status field with value “offline” for reported values.



```python
def create_temp_shadow_report(temperature):
    return ujson.dumps({
        "state": {
            "reported": {
                "temperature": temperature,
                "status": "online"
            }
        }
    })
```

FIGURE 17. JSON string to update shadow

5.3 Core device program

5.3.1 Overview

The main function of the core device is to handle communication with the cloud and respond to events from the ESP32 microcontroller. To provide visual feedback on the state of the program, a 7-segment display and LEDs were integrated into the project. LEDs indicate whether the client device is connected or not, and whether the temperature value received from ESP32 is below 0 degrees. The 7-segment display shows the exact temperature and current time at controllable intervals. These indicators provide a quick and easy way for users to understand the current state of the system. In addition, two push buttons were added to the device to enable users to control the display's interval state. In the following chapters, the components that define the functionality of the program are explained. Figure 18 displays a picture of the system.

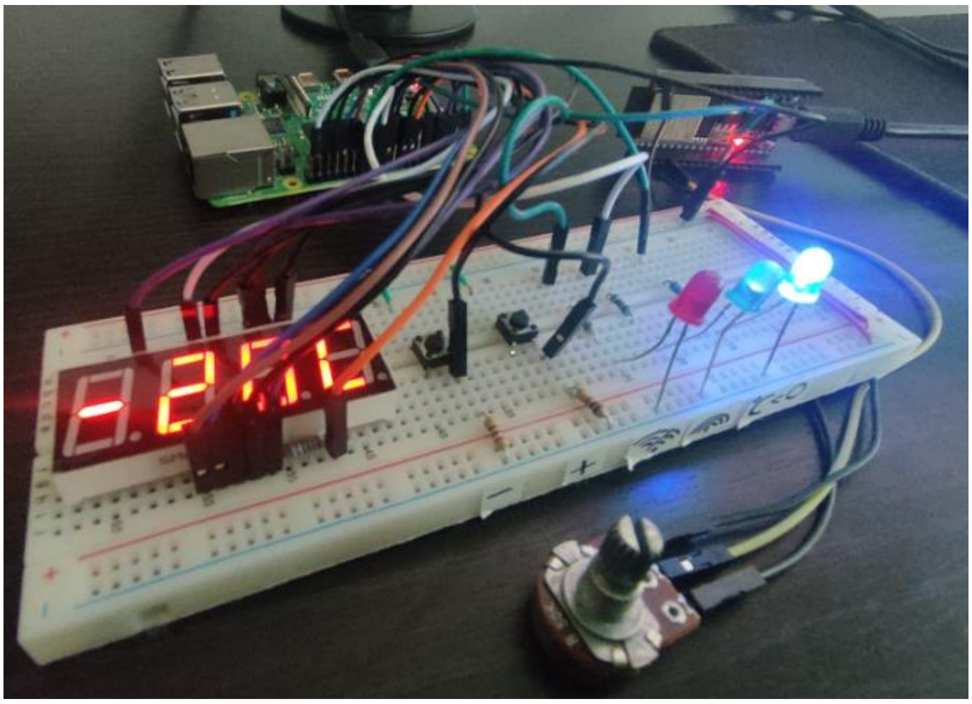


FIGURE 18. Demo application

5.3.2 Display component

A display component handles all the displayable devices connected to the core device. 4 digit 7-segment screen displays temperature data and time. LEDs are indicators of the client device's state. A green LED emits light when the client device is connected, a red LED emits light when the client device is not connected, and a blue LED emits light when the temperature is less than zero (0) degrees Celsius. The flowchart of the display component is presented in Figure 19 below.

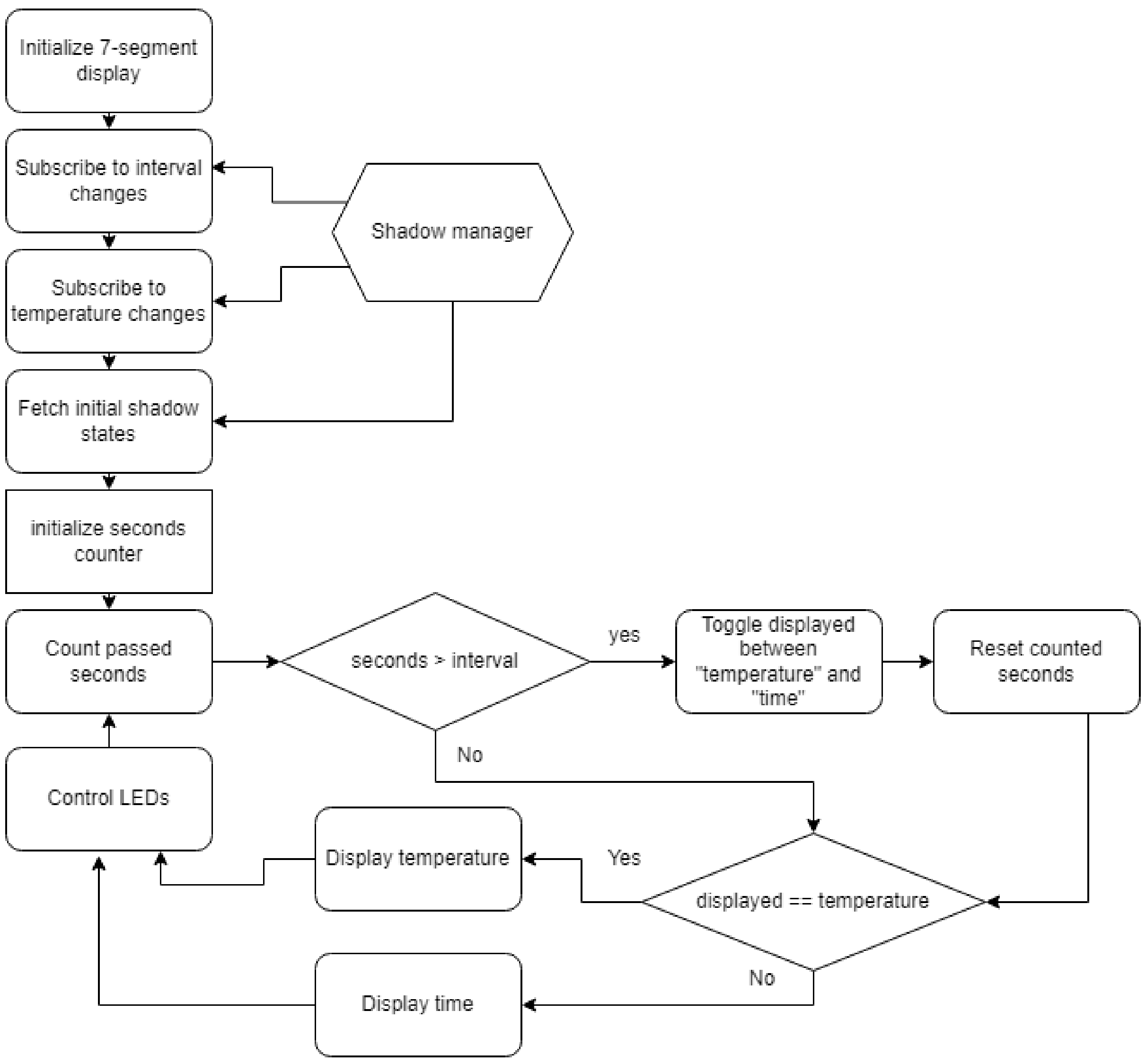
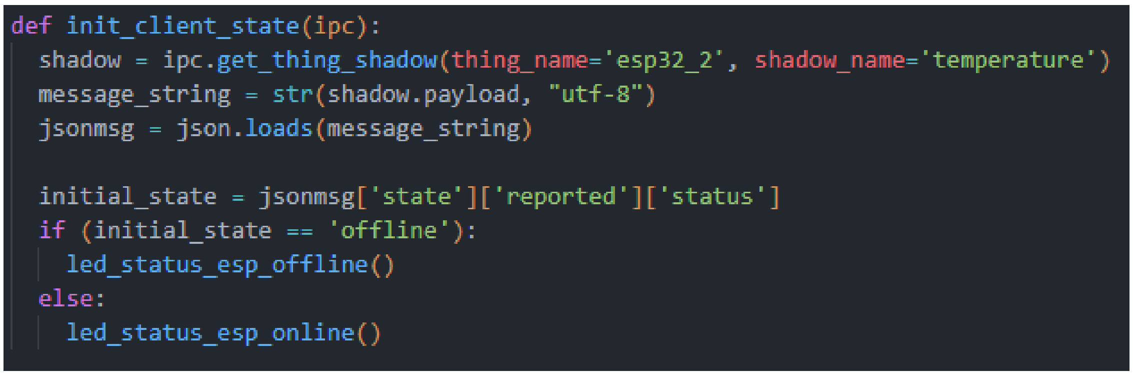

Initialize 7-segment display
Subscribe to interval changes
Subscribe to temperature changes
Fetch initial shadow states
initialize seconds counter
Count passed seconds
Control LEDs
Shadow manager
seconds > interval
yes
No
Toggle displayed between "temperature" and "time"
Reset counted seconds
Display temperature
Yes
displayed == temperature
No
Display time

FIGURE 19. Display component flowchart

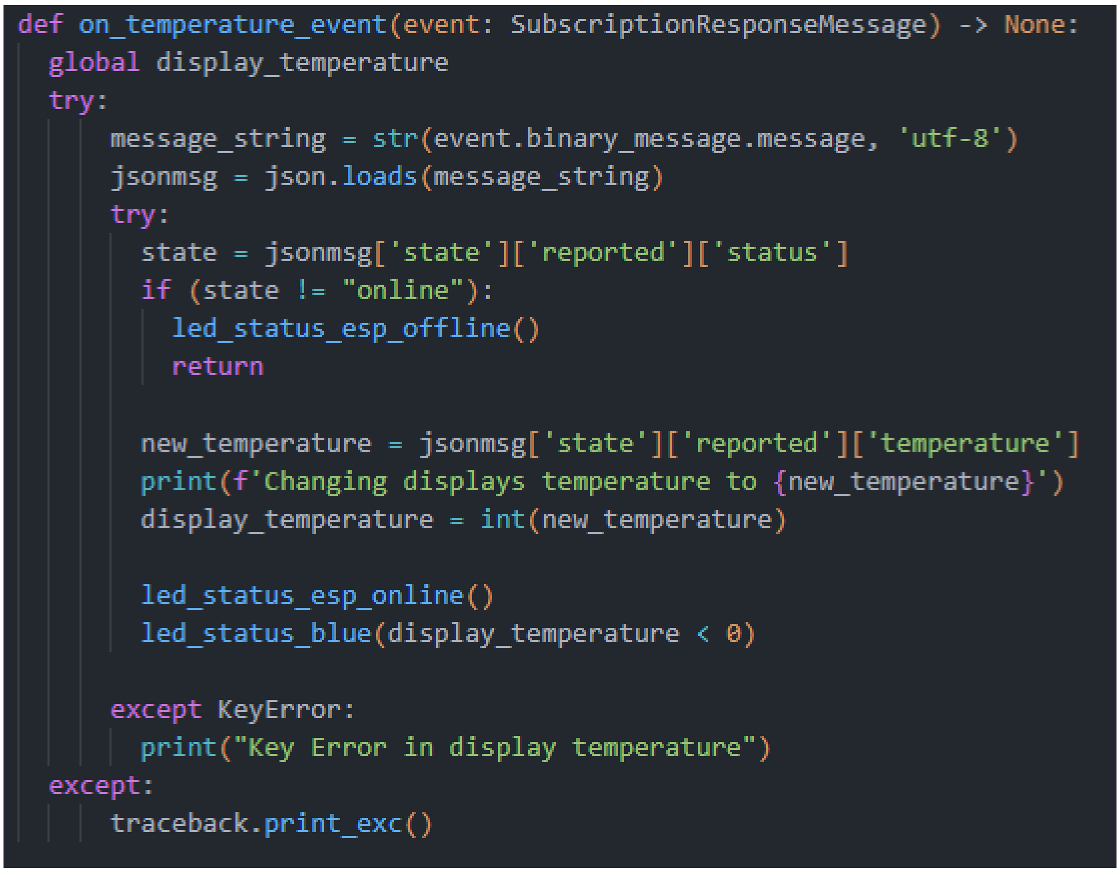
The displayable temperature and interval values of the component are obtained from the shadow manager. When the program first runs, shadow states are initialized by using the "get_thing_shadow" function from the GreengrassCoreIPCClientV2 library (see Figure 20).



```
def init_client_state(ipc):
    shadow = ipc.get_thing_shadow(thing_name='esp32_2', shadow_name='temperature')
    message_string = str(shadow.payload, "utf-8")
    jsonmsg = json.loads(message_string)

    initial_state = jsonmsg['state']['reported']['status']
    if (initial_state == 'offline'):
        led_status_esp_offline()
    else:
        led_status_esp_online()
```

FIGURE 20. Program initializing shadow state for client connectivity status

After initialization, the displayable temperature and interval values are updated by subscribing to the MQTT topics that publish updated shadow states (see Figure 21). This allows the display component to stay up-to-date with the latest temperature and interval values, and to display accurate information on the 7-segment screen and LED indicators.

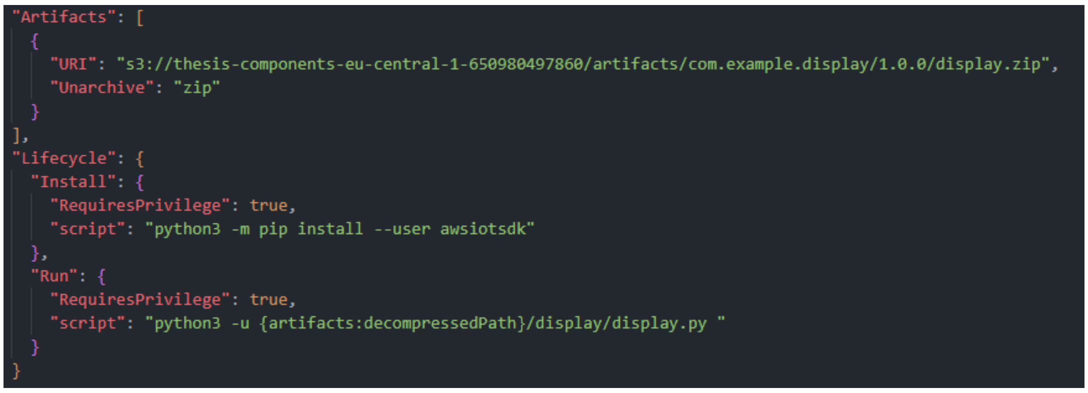


```python
def on_temperature_event(event: SubscriptionResponseMessage) -> None:
    global display_temperature
    try:
        message_string = str(event.binary_message.message, 'utf-8')
        jsonmsg = json.loads(message_string)
        try:
            state = jsonmsg['state']['reported']['status']
            if (state != "online"):
                led_status_esp_offline()
                return

            new_temperature = jsonmsg['state']['reported']['temperature']
            print(f'Changing displays temperature to {new_temperature}')
            display_temperature = int(new_temperature)

            led_status_esp_online()
            led_status_blue(display_temperature < 0)

        except KeyError:
            print("Key Error in display temperature")
    except:
        traceback.print_exc()
```

FIGURE 21. Callback function for subscribed temperature event

The display component was manually published to the Greengrass Cloud service to avoid resetting recipe policies. The display program consists of two files: one containing the function to control the 7-segment display, and the other handling all other tasks. To package these artifact files, a zip program was used to create a zip package, which was then uploaded to S3 storage.

The component's recipe file was modified so that the artifact path would point to S3 storage and unarchive was set to "zip", so the program would automatically unzip the folder on deployment. When components are published using the GDK CLI, changes to artifact paths are handled automatically, but it is possible to do it manually as well. As the documentation for extracted archives (Component recipe, 2023) state, "{artifacts:decompressedPath}" can be used to point to files inside the extracted archive artifacts folder. As files were packaged into a zip folder named "display", it was appended to the decompressed path. Figure 22 below displays the final recipe, which shows the S3 destination as well as the run command that uses the decompressed path.

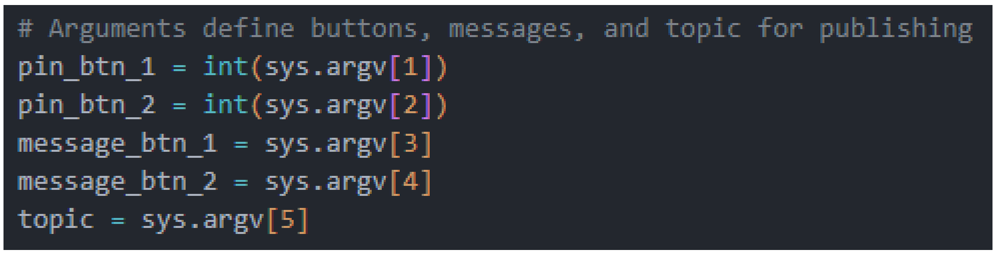


```json
"Artifacts": [
  {
    "URI": "s3://thesis-components-eu-central-1-650980497860/artifacts/com.example.display/1.0.0/display.zip",
    "Unarchive": "zip"
  }
],
"Lifecycle": {
  "Install": {
    "RequiresPrivilege": true,
    "script": "python3 -m pip install --user awsiotsdk"
  },
  "Run": {
    "RequiresPrivilege": true,
    "script": "python3 -u {artifacts:decompressedPath}/display/display.py "
  }
}
```

FIGURE 22. Recipe defines artifact folder and archive method

5.3.3 Buttons component

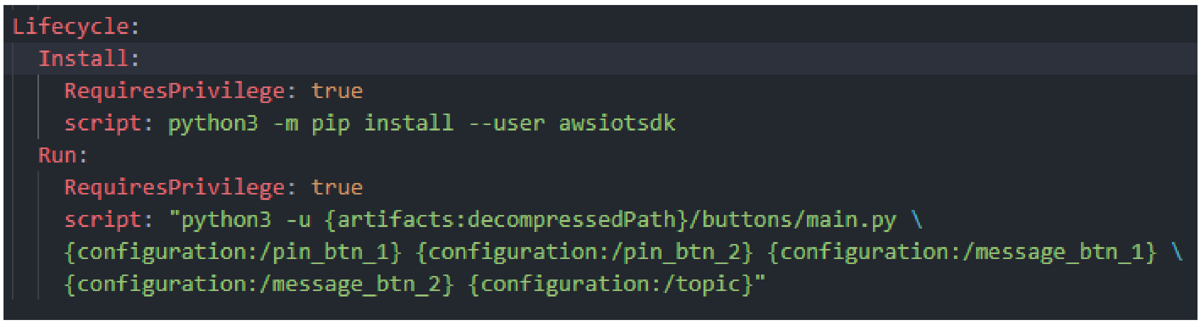
The buttons component utilizes Greengrass interprocess communication (IPC) to publish messages on a given topic. Button pins, messages, and the topic are given as program arguments, which the program then reads into variables (see Figure 23). This way, the component can be configured to work in different scenarios where two buttons are needed to control a system.



```
# Arguments define buttons, messages, and topic for publishing
pin_btn_1 = int(sys.argv[1])
pin_btn_2 = int(sys.argv[2])
message_btn_1 = sys.argv[3]
message_btn_2 = sys.argv[4]
topic = sys.argv[5]
```

FIGURE 23. Arguments define buttons, messages, and topic for publishing

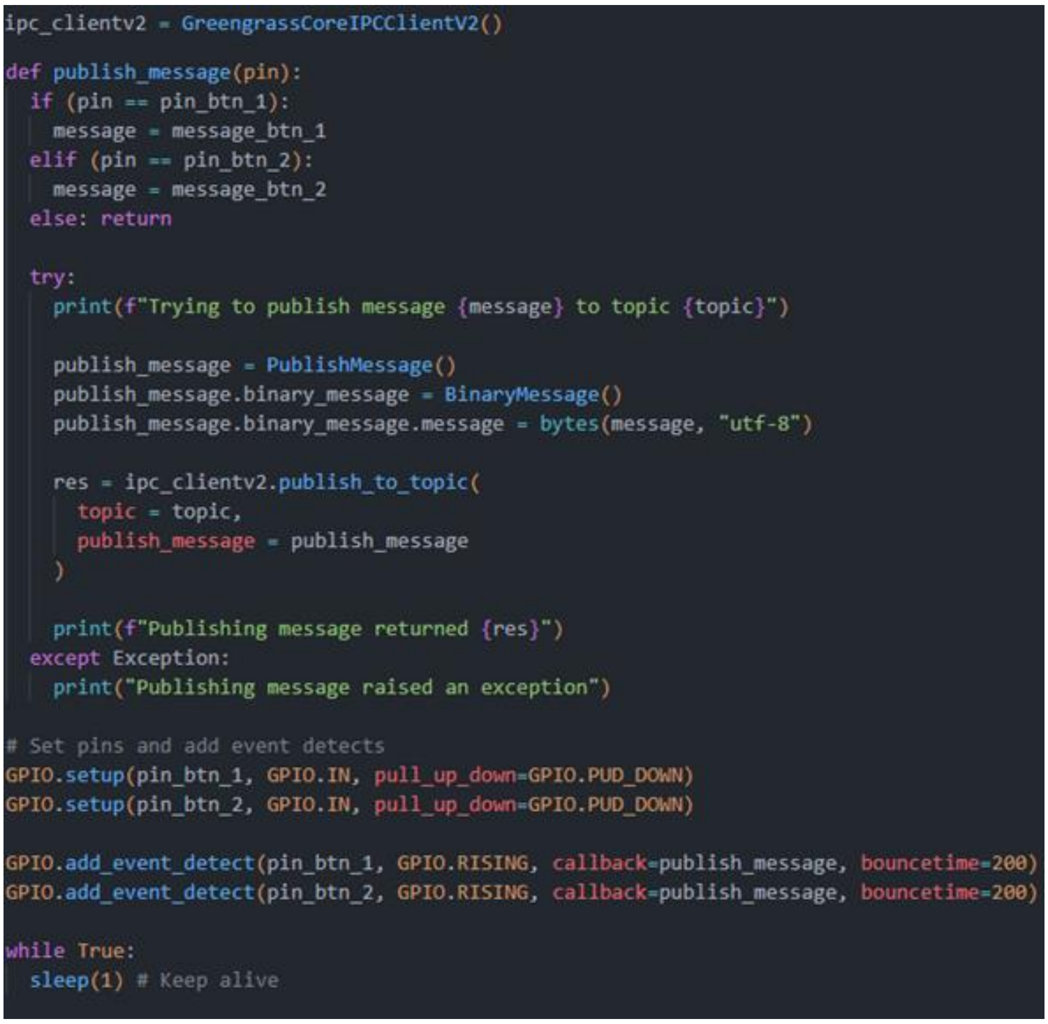
The component's recipe is used to define these given arguments, and they are given to the program in its lifecycle's run command, as shown in Figure 24. The figure also shows how the recipe has RequiresPrivilege set to true to run the script as root. This gives the component access to the Raspberry Pi's GPIO interface.



```
Lifecycle:
  Install:
    RequiresPrivilege: true
    script: python3 -m pip install --user awsiotsdk
  Run:
    RequiresPrivilege: true
    script: "python3 -u {artifacts:decompressedPath}/buttons/main.py \
    {configuration:/pin_btn_1} {configuration:/pin_btn_2} {configuration:/message_btn_1} \
    {configuration:/message_btn_2} {configuration:/topic}"
```

FIGURE 24. Recipe's lifecycle

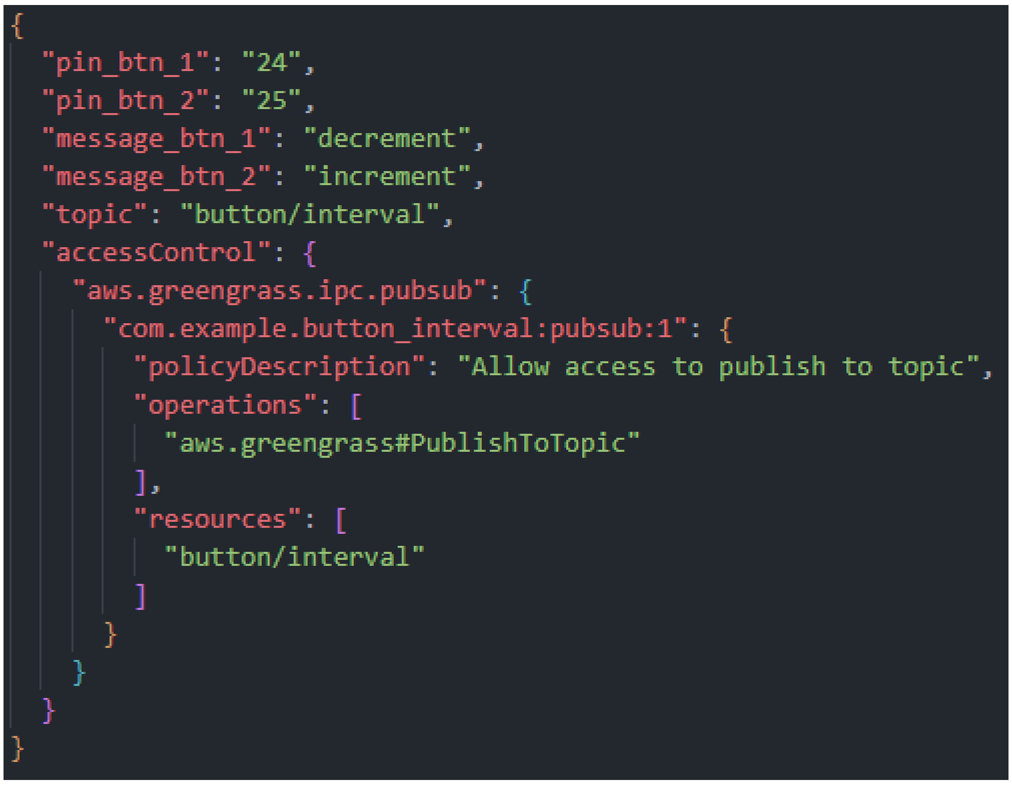
The program adds event listeners to the given button pins and publishes the message to the given topic when the button is pressed. Messages are published using the GreengrassCoreIPCClientV2 (SDK V2, 2023), as it reduces the amount of code that needs to be written to perform IPC operations (IPC, 2023). Figure 25 below shows the function that publishes a message depending on the pin received as an argument. Event detectors have this function as a callback and the pin that fires the event is sent as an argument to the function.



```python
ipc_clientv2 = GreengrassCoreIPCClientV2()

def publish_message(pin):
  if (pin == pin_btn_1):
    message = message_btn_1
  elif (pin == pin_btn_2):
    message = message_btn_2
  else: return

  try:
    print(f"Trying to publish message {message} to topic {topic}")

    publish_message = PublishMessage()
    publish_message.binary_message = BinaryMessage()
    publish_message.binary_message.message = bytes(message, "utf-8")

    res = ipc_clientv2.publish_to_topic(
      topic = topic,
      publish_message = publish_message
    )

    print(f"Publishing message returned {res}")
  except Exception:
    print("Publishing message raised an exception")

# Set pins and add event detects
GPIO.setup(pin_btn_1, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(pin_btn_2, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

GPIO.add_event_detect(pin_btn_1, GPIO.RISING, callback=publish_message, bouncetime=200)
GPIO.add_event_detect(pin_btn_2, GPIO.RISING, callback=publish_message, bouncetime=200)

while True:
  sleep(1) # Keep alive
```

FIGURE 25. Publishing message on button press

The component was published to the Greengrass Cloud service using the GDK CLI's publish command, making it possible to be deployed to any amount of core devices with the desired configuration. The component was then deployed to the Raspberry Pi with the configuration shown in Figure 26. Pins 24 and 25 were chosen to send the messages "decrement" and "increment" respectively to the topic "button/interval". Component was given permission to publish on this topic.



```
{
  "pin_btn_1": "24",
  "pin_btn_2": "25",
  "message_btn_1": "decrement",
  "message_btn_2": "increment",
  "topic": "button/interval",
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.button_interval:pubsub:1": {
        "policyDescription": "Allow access to publish to topic",
        "operations": [
          "aws.greengrass#PublishToTopic"
        ],
        "resources": [
          "button/interval"
        ]
      }
    }
  }
}
```

FIGURE 26. Buttons component configuration

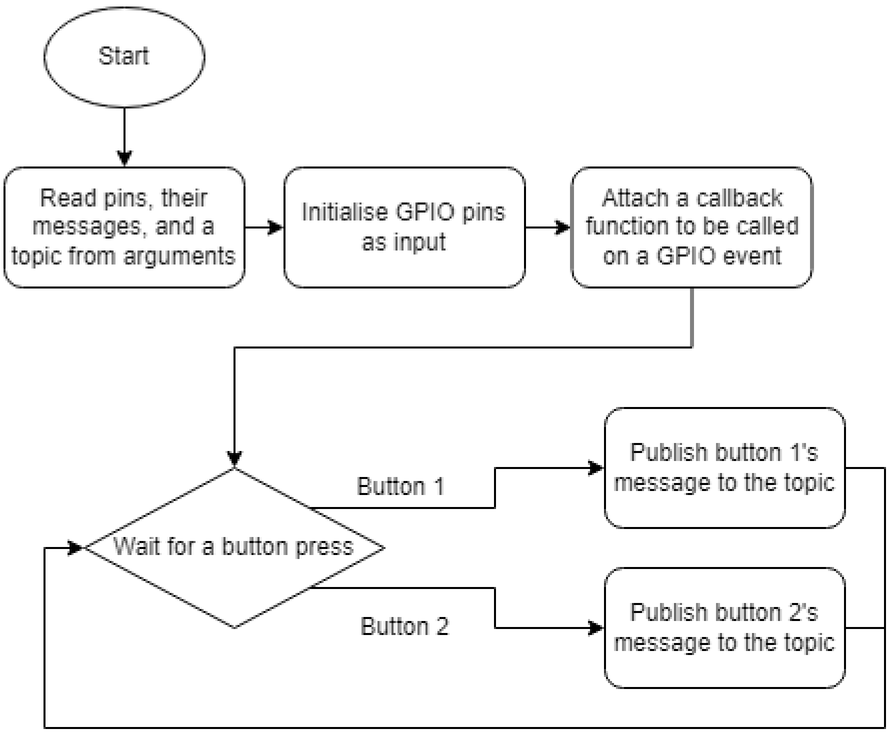

Start
Read pins, their messages, and a topic from arguments
Initialise GPIO pins as input
Attach a callback function to be called on a GPIO event
Wait for a button press
Button 1
Button 2
Publish button 1's message to the topic
Publish button 2's message to the topic

FIGURE 27. Flowchart for button component

5.3.4 Shadow interval component

The shadow interval component is designed to demonstrate interprocess communication between two different components and to show how shadow state can be kept up-to-date when it is changed from multiple interfaces. The component reacts to messages from both the physical interface (GPIO) and the remote cloud via MQTT topics that notify shadow updates.

To keep itself up-to-date with the shadow value of the interval, the component initializes a local variable with the value from the shadow manager when the program first runs. It then listens to messages from the button component, and upon receiving a message, updates the local variable and updates the shadow by setting the desired field of the shadow to match the local variable.

The component also subscribes to a specific MQTT topic that publishes messages when the shadow's desired and reported values differ. The button component triggers this MQTT message to be published, and if the desired state and local value are up-to-date, it is reported back unless the service requires it to be desired again. This process continues until the program settles and agrees to have the local variable with the value of desired, reporting it back accordingly. Similarly, this MQTT topic publishes a message when the shadow is updated from the AWS console, and the device follows the same procedure to keep its state synced with the shadow manager.

Figure 28 below displays program's flowchart.

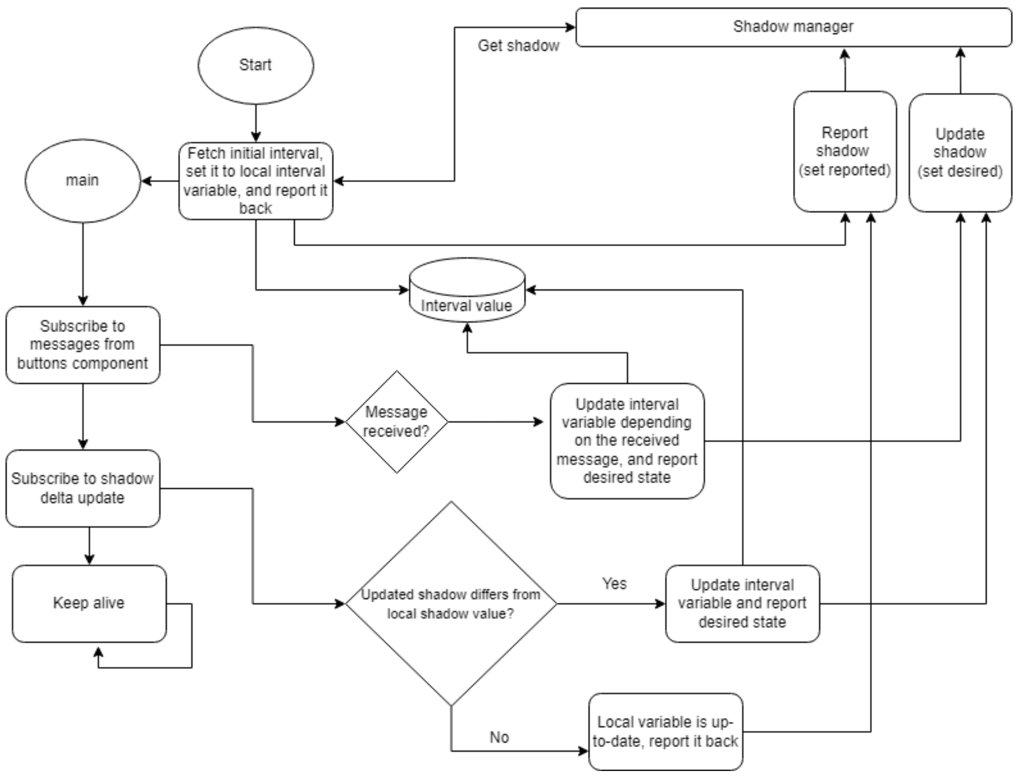
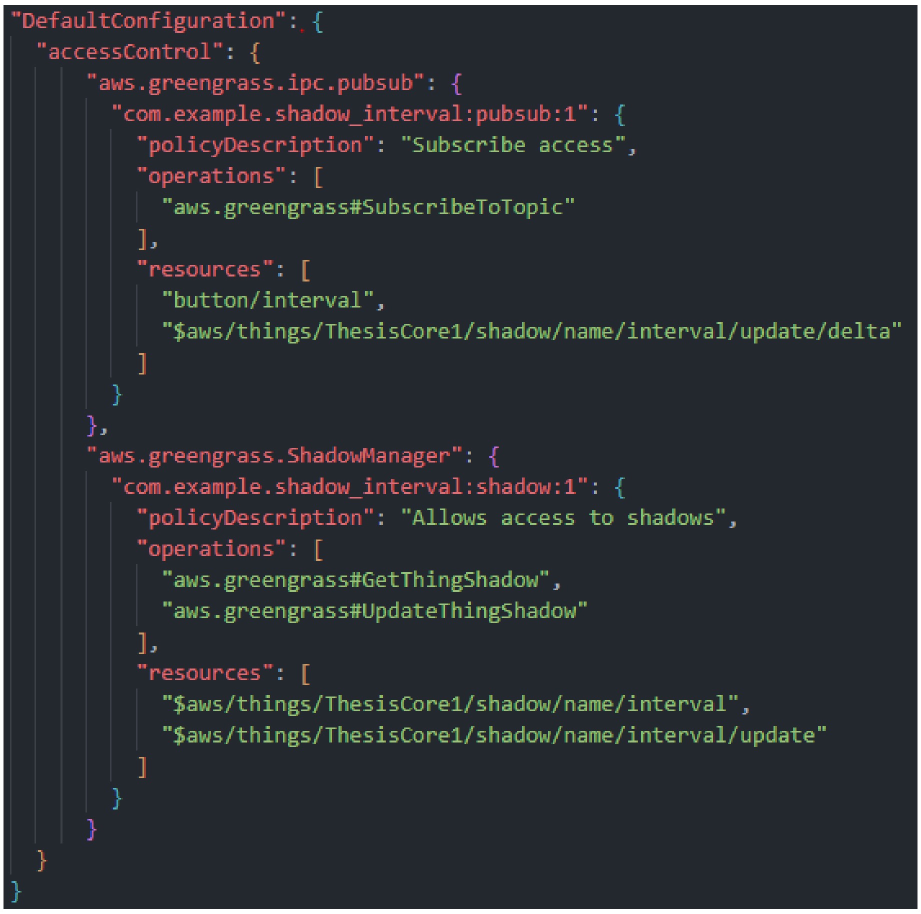

Shadow manager
Get shadow
Start
Report shadow (set reported)
Update shadow (set desired)
main
Fetch initial interval, set it to local interval variable, and report it back
Interval value
Subscribe to messages from buttons component
Message received?
Update interval variable depending on the received message, and report desired state
Subscribe to shadow delta update
Keep alive
Updated shadow differs from local shadow value?
Yes
Update interval variable and report desired state
No
Local variable is up-to-date, report it back

FIGURE 28. Shadow interval flowchart

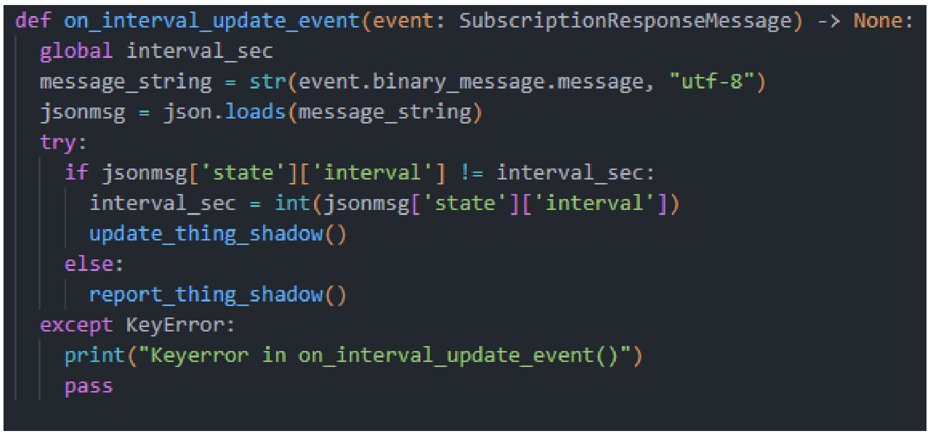
To interact with the shadow manager and perform interprocess communication, the component requires proper recipe configuration (see Figure 29). The recipe configuration specifies the access control policies that allow the component to access the necessary topics and perform the required operations to keep itself updated. In this case, the access control policies were set to enable the component to subscribe to the topic on which the button component publishes its messages and where the shadow update is published. Additionally, the component was given access to the shadow manager with the ability to perform operations such as getting and updating the shadow.



```
"DefaultConfiguration": {
  "accessControl": {
      "aws.greengrass.ipc.pubsub": {
        "com.example.shadow_interval:pubsub:1": {
          "policyDescription": "Subscribe access",
          "operations": [
            "aws.greengrass#SubscribeToTopic"
          ],
          "resources": [
            "button/interval",
            "$aws/things/ThesisCore1/shadow/name/interval/update/delta"
          ]
        }
      },
      "aws.greengrass.ShadowManager": {
        "com.example.shadow_interval:shadow:1": {
          "policyDescription": "Allows access to shadows",
          "operations": [
            "aws.greengrass#GetThingShadow",
            "aws.greengrass#UpdateThingShadow"
          ],
          "resources": [
            "$aws/things/ThesisCore1/shadow/name/interval",
            "$aws/things/ThesisCore1/shadow/name/interval/update"
          ]
        }
      }
  }
}
```

FIGURE 29. Recipe defines component's access control policies

The callback function for the interval update event is shown in Figure 30. The function compares the received shadow value with the local interval value and calls the function to update the shadow until the local variable matches the received value and is then reported back.



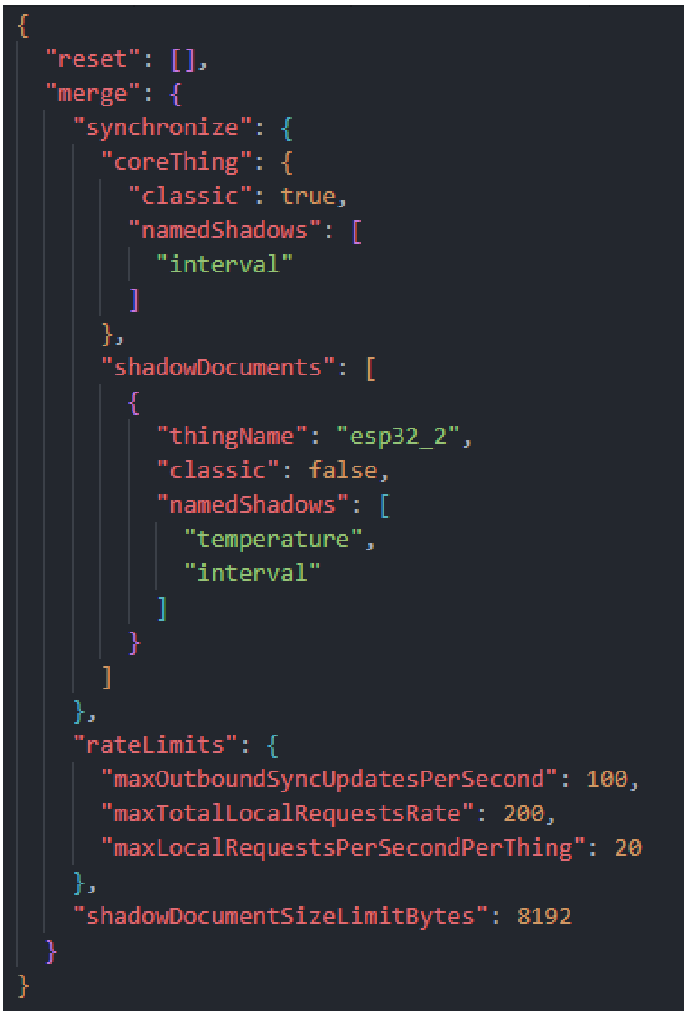
```
def on_interval_update_event(event: SubscriptionResponseMessage) -> None:
  global interval_sec
  message_string = str(event.binary_message.message, "utf-8")
  jsonmsg = json.loads(message_string)
  try:
    if jsonmsg['state']['interval'] != interval_sec:
      interval_sec = int(jsonmsg['state']['interval'])
      update_thing_shadow()
    else:
      report_thing_shadow()
  except KeyError:
    print("Keyerror in on_interval_update_event()")
    pass
```

FIGURE 30. Local variable tracks the value in shadow

5.3.5 Shadow manager

Shadow manager is an AWS-provided component that enables local shadow server on core device and synchronizes local device shadow with AWS IoT Shadow service (Shadow manager, 2023). Shadow Manager component configuration is used to define which shadows are synced, as well as to set parameters like interval rates and maximum synchronization requests per second (see Figure 31). For the program made for the Thesis, core devices display interval shadow along with client devices temperature and measurement interval shadows were set to be synchronized with the cloud at real time interval.

Each shadow has a reserved MQTT topic and a HTTP URL for creating, updating, and deleting a shadow. For this project MQTT topics were used to modify and read the shadow states. Each shadow has a MQTT topic that is appended with get, update, and delete to define the operation. For example, to update a shadow a MQTT publish message is sent to /update path. Another way to update the shadow is to use GreengrassCoreIPCV2 library and the functions it has for accessing and managing device stare.



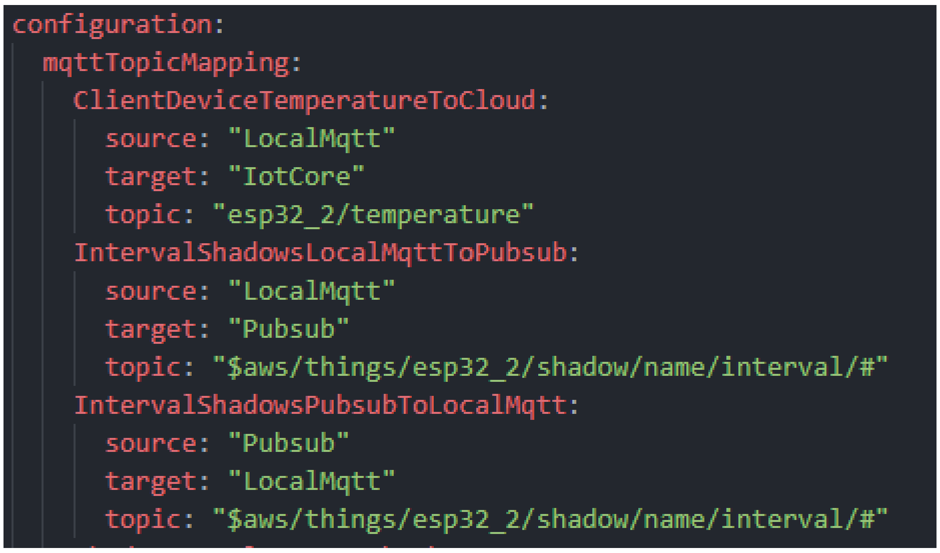
```
{
  "reset": [],
  "merge": {
    "synchronize": {
      "coreThing": {
        "classic": true,
        "namedShadows": [
          "interval"
        ]
      },
      "shadowDocuments": [
        {
          "thingName": "esp32_2",
          "classic": false,
          "namedShadows": [
            "temperature",
            "interval"
          ]
        }
      ]
    },
    "rateLimits": {
      "maxOutboundSyncUpdatesPerSecond": 100,
      "maxTotalLocalRequestsRate": 200,
      "maxLocalRequestsPerSecondPerThing": 20
    },
    "shadowDocumentSizeLimitBytes": 8192
  }
}
```

FIGURE 31. Shadow manager configuration defines the shadows to be synced with IoT Core

5.3.6 MQTT communication

MQTT Moquette (Moquette, 2023) is an AWS-provided component that handles the MQTT messaging between core device and client devices. Moquette depends on another AWS-provided component that handles authorization of the client devices (Auth, 2023). Together these components deployed to the core device allow client devices to connect to it securely.

MQTT Bridge is an AWS-provided component that acts as a bridge between different MQTT communication brokers such as Local publish/subscribe (pubsub), Local MQTT, and IoT Core (see Figure 33). Component's recipe defines mapping between these brokers with rules. As an example, a message from client device comes through Moquette's Local MQTT broker, for this message to be used by components such as a Shadow Manager that operates behind the pubsub broker (Shadow manager, 2023), the message must be bridged between these two brokers. Another example is a message from a client device that is routed to IoT Core. Figure 32 below shows an example of routing configuration.



```
configuration:
  mqttTopicMapping:
    ClientDeviceTemperatureToCloud:
      source: "LocalMqtt"
      target: "IotCore"
      topic: "esp32_2/temperature"
    IntervalShadowsLocalMqttToPubsub:
      source: "LocalMqtt"
      target: "Pubsub"
      topic: "$aws/things/esp32_2/shadow/name/interval/#"
    IntervalShadowsPubsubToLocalMqtt:
      source: "Pubsub"
      target: "LocalMqtt"
      topic: "$aws/things/esp32_2/shadow/name/interval/#"
```

FIGURE 32. MQTT Bridge configuration

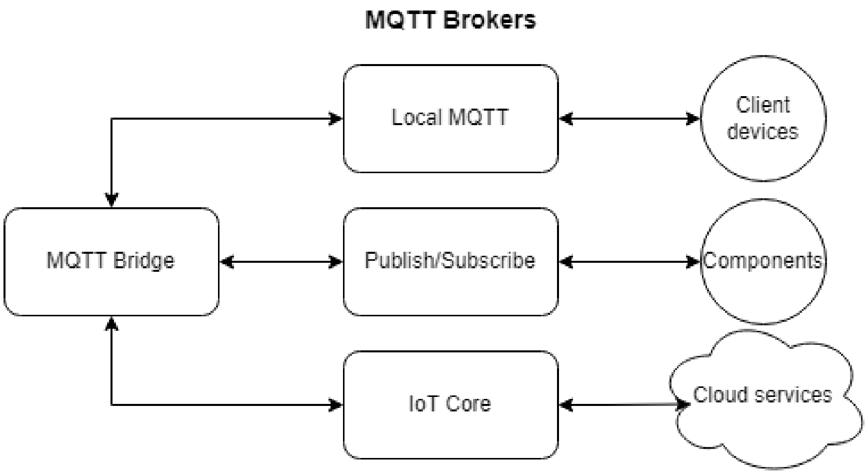


FIGURE 33. MQTT Bridge routes messages between different brokers

5.3.7 Synchronizing data after network disruptions

Edge computing is designed to provide fast responses to local events by minimizing the need for cloud computing. However, certain tasks, such as data analysis, can benefit from the scalability and resources provided by the cloud. To enable effective data analysis, data gathered by edge computing needs to be seamlessly transferred to the cloud, even in the event of network disruptions. This is why data synchronization after network disruptions was added to the program.

Greengrass nucleus component that is the minimum requirement to run the AWS IoT Greengrass has an option for spooler configuration, that adds in-memory storage for undelivered MQTT messages. The spooler component can be configured to set a maximum size for the cache and uses first-in, first-out cache replacement when the cache is full. The spooler has also a configuration option to store MQTT messages that are sent with Quality of Service level of 0 (QoS 0). QoS 0 is an agreement between a MQTT sender and recipient that messages are tried to be delivered only once and the recipient does not acknowledge the sender on successful delivery. In a situation where the core device loses connection to cloud, but retains the connection with the ESP32 device, the messages from ESP32 are stored in the spooler and resend after network reconnects. However, this configuration only affects the connection between the core device and cloud. To successfully deliver every message from ESP32, a QoS with a level of 1 for at-least-once or level 2 for exactly once delivery would be needed to be used.

For example, as a message with QoS 0 does not have a guarantee for a delivery and would get discarded unless spooler was configured to retain it. The spooler configuration was set to keep the messages with QoS 0 and the cache’s max size was defined to be approximately 10.4 megabytes (see Figure 34).

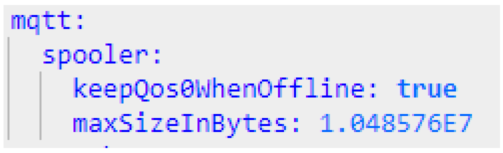

```
mqtt:
  spooler:
    keepQos0WhenOffline: true
    maxSizeInBytes: 1.048576E7
```

FIGURE 34. Spooler configuration

As of April 2023, the AWS Greengrass nucleus component's GitHub repository has an active issue (Spooler issue, 2023) regarding the lack of persistent storage for the spooler used by the MQTT client. Currently, the spooler only provides in-memory storage, which means that the message queue will be cleared when the device shuts down. Fortunately, the AWS team has recognized this issue and is planning to implement a persistent storage solution for the spooler in the coming months. As of now another option would be to use Stream manager component (chapter 3.3.3).

5.3.8 Network time procotol server

AWS Greengrass does not have a built-in component that provides time synchronization between client and core devices. However, an effective method for synchronizing time between devices is to use Network Time Protocol (NTP) servers. NTP servers synchronize time between a client and server. A NTP server was installed to the core device using apt package manager. With NTP server running in the core device, ESP32 can request the time from it to update its clock.

5.4 IoT analytics

5.4.1 Overview

AWS IoT analytics (Analytics) is a service for storing and analyzing large amounts of IoT data. With its tools to manipulate the data, Analytics helps to address some issues the raw IoT data might have such as corrupted messages or gaps in the data. Processed data is stored into a time-series data store for faster retrieval and analysis. Stored data can be analyzed by running queries using the built-in SQL query engine or perform more complex analytics and machine learning inference using Jupyter Notebook. (Analytics, 2023)

Temperature data from ESP32 microcontroller was forwarded to the Analytics service to demonstrate how data from edge device can be transmitted to cloud to perform analysis on the data.

5.4.2 Architecture

Analytics service consists of channels, pipelines, data stores, data sets, and notebooks (see Figure 35). Channels are entry points for data ingestion that pull messages or data from other AWS sources, such as MQTT topics or IoT Core. A pipeline then consumes messages from one or more channels with option to manipulate the data before storing them to data stores. Data sets are created by querying the data store, which can then be analyzed by notebooks, or viewed using a service like QuickSight.

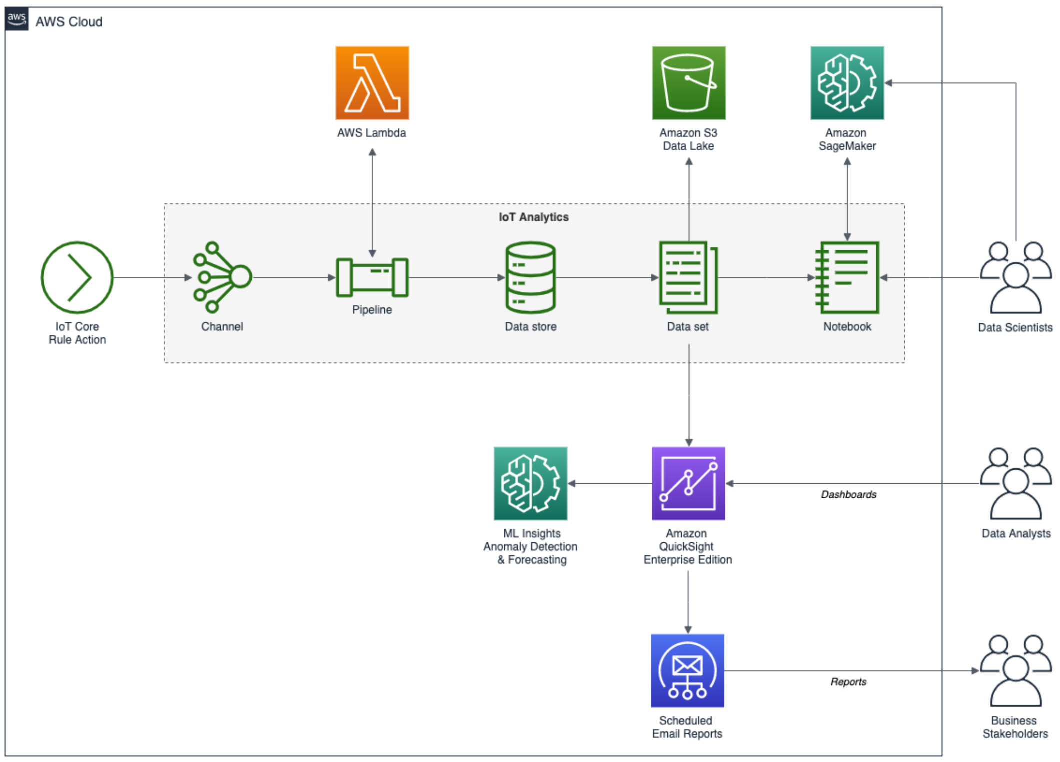
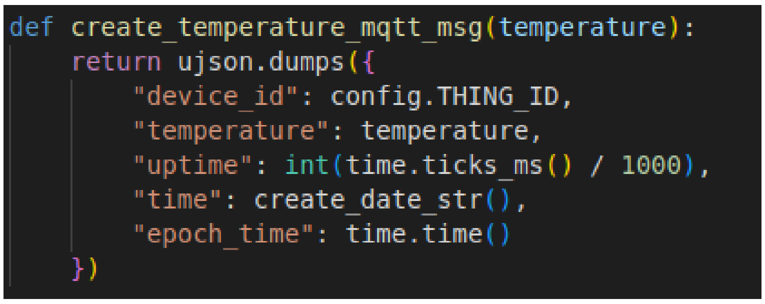


FIGURE 35. AWS IoT Analytics architecture

5.4.3 Using Analytics

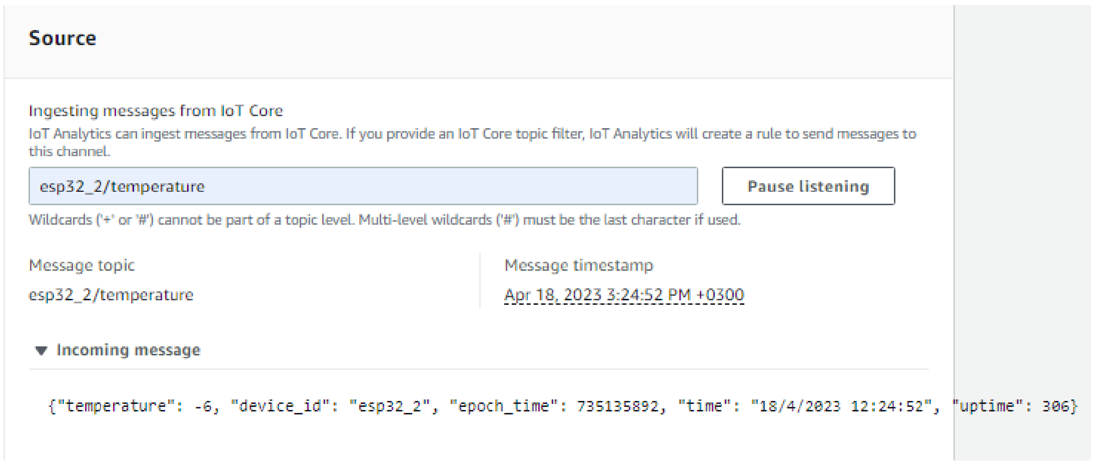
ESP32 microcontroller sends temperature data to a specific MQTT topic the core device routes to IoT Core. The message is in a JSON format and consists of fields for temperature, measurement time, device id, and device uptime (see Figure 36).



```python
def create_temperature_mqtt_msg(temperature):
    return ujson.dumps({
        "device_id": config.THING_ID,
        "temperature": temperature,
        "uptime": int(time.ticks_ms() / 1000),
        "time": create_date_str(),
        "epoch_time": time.time()
    })
```

FIGURE 36. Temperature data from ESP32

First, a channel was created to subscribe to the MQTT topic to ingest the message into Analytics service. Channel stores unprocessed messages in a S3 storage and the retention period along with the S3 location can be configured. For this project the data retention was set to 1 day and S3 storage was set to be automatically managed by the Analytics service. During channel configuration the subscribed MQTT topic shows the message received from ESP32 contains the same fields as the message that was sent (see Figure 37).



Source

Ingesting messages from IoT Core

IoT Analytics can ingest messages from IoT Core. If you provide an IoT Core topic filter, IoT Analytics will create a rule to send messages to this channel.

esp32_2/temperature

Wildcards ('+' or '#') cannot be part of a topic level. Multi-level wildcards ('#') must be the last character if used.

Message topic: esp32_2/temperature

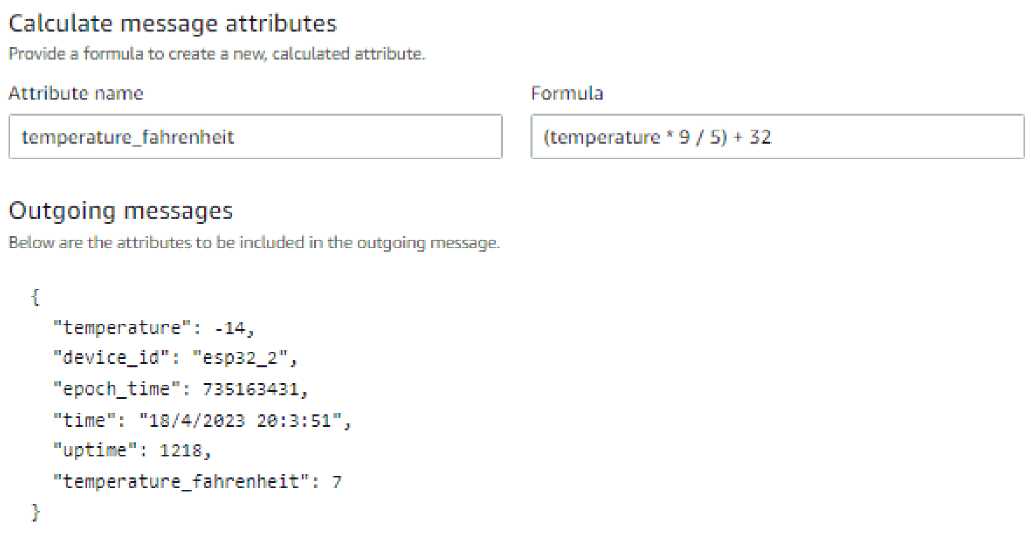
Message timestamp: Apr 18, 2023 3:24:52 PM +0300

Incoming message

```
{"temperature": -6, "device_id": "esp32_2", "epoch_time": 735135892, "time": "18/4/2023 12:24:52", "uptime": 306}
```

FIGURE 37. Channel ingesting MQTT message

After successfully setting up the channel, a pipeline was created to consume messages from this channel. Pipeline has tools to process the data, such as removing, or renaming fields, or adding new fields by performing mathematical transformations on the data. A simple mathematical activity was added to transform the data to contain a field for temperature in Fahrenheit. Finally, the pipeline was set to store the processed message into data store. Figure 38 below shows the processed message that contains the new calculated value for Fahrenheit.


Calculate message attributes

Provide a formula to create a new, calculated attribute.

Attribute name	Formula
temperature_fahrenheit	(temperature * 9 / 5) + 32

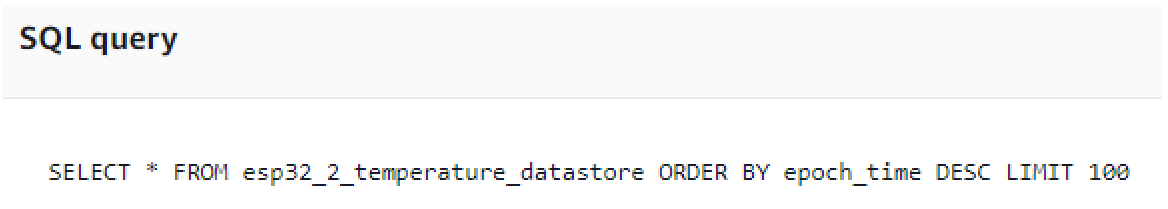
Outgoing messages

Below are the attributes to be included in the outgoing message.

```
{
  "temperature": -14,
  "device_id": "esp32_2",
  "epoch_time": 735163431,
  "time": "18/4/2023 20:3:51",
  "uptime": 1218,
  "temperature_fahrenheit": 7
}
```

FIGURE 38. Pipeline transforms the data with Fahrenheit conversion

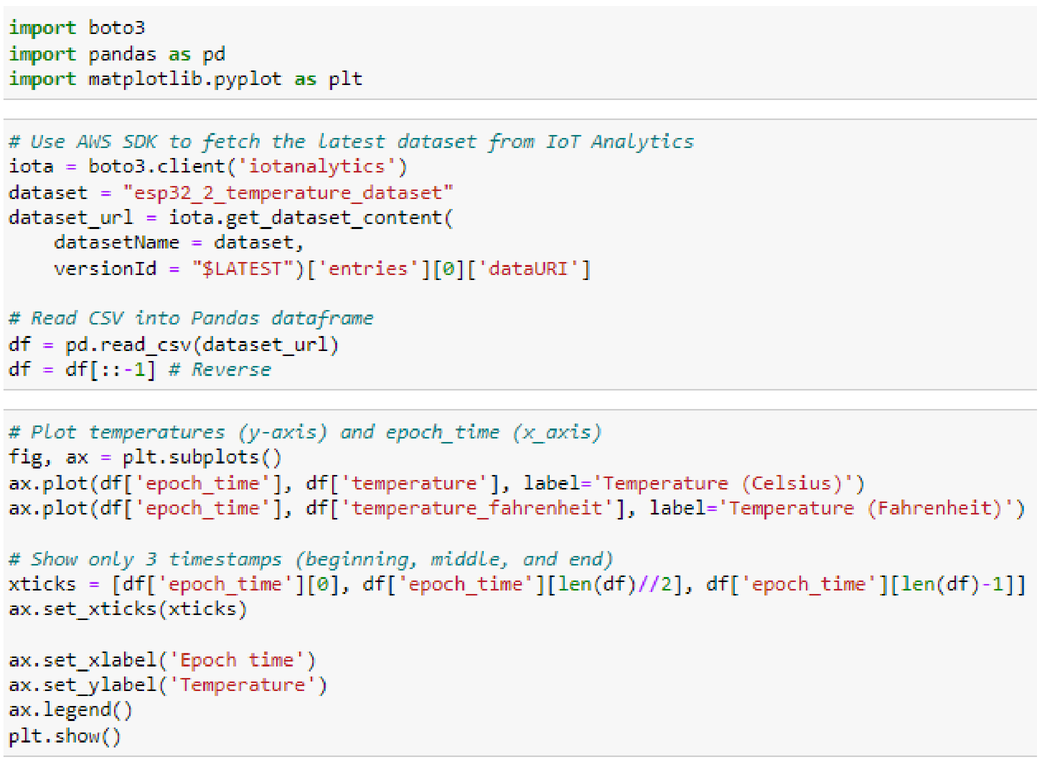
Data stores store processed messages from pipelines and use S3 for storage similarly as channels, allowing users to determine the data retention and handling policies. Data store is a scalable and queryable repository of messages, that is queried to create a dataset. A dataset contains a portion of the data store to be used for analysis. A dataset can be triggered to query a data store in time intervals, such as every 5 minutes, or run manually. A SQL query to fetch the last 100 items ordered by the epoch time was done to create a dataset (see Figure 39).


SQL query

```
SELECT * FROM esp32_2_temperature_datastore ORDER BY epoch_time DESC LIMIT 100
```

FIGURE 39. SQL query for creating a dataset

Finally, the temperature data from the dataset was plotted using a Jupyter Notebook instance. Jupyter Notebooks are capable of performing machine learning and statistical analysis on the data. While it was not within the scope of this thesis to perform such analysis, the data was only plotted to demonstrate successful data forwarding between different AWS services. The notebook was created using the AWS Console, and the necessary IAM policies to grant access to the S3 storage containing the datasets were created during its creation. The code utilized the AWS SDK (see Figure 40) to fetch the latest dataset from IoT Analytics and was then read into a Pandas dataframe for easier access to the different data fields. The plot was created using Matplotlib, a powerful visualization tool for Python3. The plot displayed the temperature data in Celsius, as well as in Fahrenheit, which was added to the data during the pipeline (see Figure 41).



```
import boto3
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Use AWS SDK to fetch the latest dataset from IoT Analytics
iota = boto3.client('iotanalytics')
dataset = "esp32_2_temperature_dataset"
dataset_url = iota.get_dataset_content(
    datasetName = dataset,
    versionId = "$LATEST")['entries'][0]['dataURI']

# Read CSV into Pandas dataframe
df = pd.read_csv(dataset_url)
df = df[::-1] # Reverse
```

```
# Plot temperatures (y-axis) and epoch_time (x_axis)
fig, ax = plt.subplots()
ax.plot(df['epoch_time'], df['temperature'], label='Temperature (Celsius)')
ax.plot(df['epoch_time'], df['temperature_fahrenheit'], label='Temperature (Fahrenheit)')

# Show only 3 timestamps (beginning, middle, and end)
xticks = [df['epoch_time'][0], df['epoch_time'][len(df)//2], df['epoch_time'][len(df)-1]]
ax.set_xticks(xticks)

ax.set_xlabel('Epoch time')
ax.set_ylabel('Temperature')
ax.legend()
plt.show()
```

FIGURE 40. Python code to display data from dataset

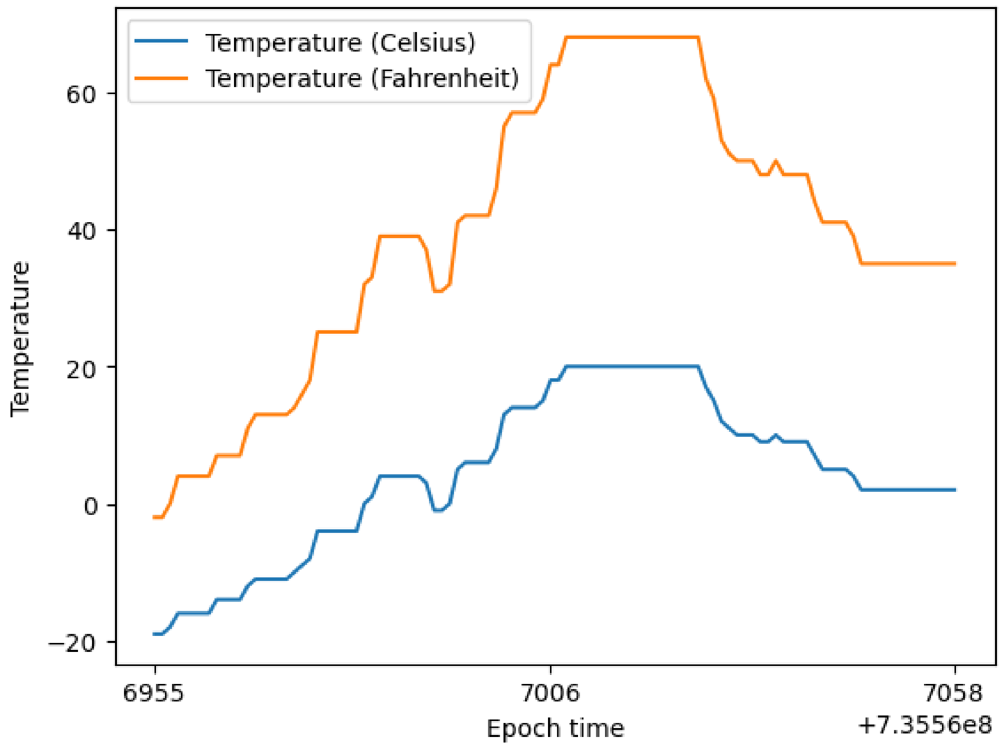


FIGURE 41. Plotted temperature data in Jupyter Notebook

6 SUMMARY

6.1 Conclusion

This thesis aimed to research AWS Greengrass and its use in developing IoT applications. To demonstrate the features of the Greengrass software, a proof-of-concept demo application was built, which utilized three layers of IoT: the sensing layer, edge layer, and cloud layer. The resulting IoT system was remotely monitorable, controllable, and upgradeable. Edge computing was emphasized by connecting the sensing and edge devices inside a local network. This demonstrated how to create a system that would work even in rural areas with low connectivity. Updates to the device state and software and the synchronization of data occurred automatically upon restoring the connection.

This thesis and demo application provides a comprehensive overview of Greengrass's capabilities in the edge computing field. The demo application proves that Greengrass and its ready-made components offer a strong foundation for IoT application development, particularly in remote monitoring and updates. These features are often the first requirements for any IoT system design, and the software's capabilities make it a valuable tool for developers facilitating the development process.

6.2 Discussion

The time I spent with Greengrass improved my knowledge of AWS services targeted to Internet of Things. Prior to this project, my experience with AWS IoT services was minimal so as with many software development projects, the initial focus was on making the different pieces work together. The first iteration of the proof-of-concept demo application was built with this mindset, and as a result, some of the decisions made regarding Greengrass architecture may have been questionable. Nonetheless, the experience gained during this process was invaluable in terms of understanding how the different components of Greengrass fit together, and the subsequent iterations could focus on refining the code and redefining the architecture.

Although the proof-of-concept demo application successfully showcased the features of Greengrass, due to the software's extensive feature list, not all its capabilities could be tested during the development. As this thesis focuses on the components and their development, future research could explore the scalability of Greengrass. For example, a study could investigate how custom Linux distributions can be built with Greengrass Core software and how fleet provisioning can be used to authenticate these devices at scale. Additionally, the Cloud discovery feature of client devices to find their core device could be examined to understand how it could automate the development process and remove the need for hard coded certificates. Such research could further expand the potential of Greengrass in IoT application development where millions of devices are needed to build smart systems.

7 REFERENCES

Angel, N. A., Ravindran, D., Vincent, P., Srinivasan, K., & Hu, Y.-C. (2021). Recent Advances in Evolving Computing Paradigms: Cloud, Edge, and Fog Technologies. doi:https://doi.org/10.3390/s22010196

AWS Greengrass. (2023). Retrieved 3 10, 2023, from AWS Website: https://aws.amazon.com/greengrass/

AWS Identity and Access Management. (2023, 3). Retrieved 3 22, 2023, from https://aws.amazon.com/iam/

AWS IoT. (2023, 3 10). Retrieved 10 3, 2023, from https://aws.amazon.com/iot/

AWS IoT Device SDK v2 for Python documentation. (2023, 4 14). Retrieved 4 14, 2023, from https://aws.github.io/aws-iot-device-sdk-python-v2/awsiot/greengrasscoreipc.html#awsiot.greengrasscoreipc.clientv2.GreengrassCoreIPCClientV2

AWS IoT Device SDKs, Mobile SDKs, and AWS IoT Device Client. (2023, 3). Retrieved 3 13, 2023, from https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html

AWS IoT Device Tester for AWS IoT Greengrass. (2023). Retrieved 3 10, 2023, from https://aws.amazon.com/greengrass/device-tester/

AWS IoT Greengrass Community Components. (2023). Retrieved 3 10, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/greengrass-software-catalog.html

AWS IoT Greengrass component recipe reference. (2023, 4 14). Retrieved 4 14, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/component-recipe-reference.html

AWS IoT Greengrass Development Kit Command-Line Interface. (2023, 4 11). Retrieved 4 11, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/greengrass-development-kit-cli.html

AWS IoT Greengrass development tools. (2023, 3). Retrieved 3 23, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/greengrass-development-tools.html

AWS IoT Greengrass Features. (2023). Retrieved 10 3, 2023, from https://aws.amazon.com/greengrass/features/

AWS IoT Greengrass ML Inference. (2023, 3). Retrieved 3 13, 2023, from https://aws.amazon.com/greengrass/ml/

AWS IoT Greengrass Public Components. (2023). Retrieved 3 10, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/public-components.html

Client device auth. (2023, 4 14). Retrieved 4 14, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/client-device-auth-component.html

Čolaković, A., & Hadžialić, M. (2018). Internet of Things (IoT): A Review of Enabling Technologies, challenges, and open research issues. *Computer Networks*. doi:https://doi.org/10.1016/j.comnet.2018.07.017

Create AWS IoT Greengrass components. (2023, 4). Retrieved 4 11, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/create-components.html

Deploy AWS IoT Greengrass components to devices. (2023, 4 25). Retrieved 4 25, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/manage-deployments.html

Develop AWS IoT Greengrass components. (2023). Retrieved 3 13, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/develop-greengrass-components.html

Device authentication and authorization for AWS IoT Greengrass. (2023, 3). Retrieved 3 22, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/device-auth.html

ESP32. (2023, 4 13). Retrieved 4 13, 2023, from https://www.espressif.com/en/products/socs/esp32

ESP32 MicroPython. (2023, 4 17). Retrieved 4 17, 2023, from https://micropython.org/download/esp32/

Esptool. (2023, 4 17). Retrieved 4 17, 2023, from https://github.com/espressif/esptool

Etteplan. (2023, 2 22). *Etteplan Oy*. Retrieved from Etteplan Oy web site: https://etteplan.com

Getting started with MicroPython on the ESP32. (2023, 4 17). Retrieved 3 17, 2023, from https://docs.micropython.org/en/latest/esp32/tutorial/intro.html

GitHub repository for AWS Geengrass Software Catalog. (2023, 3). Retrieved 3 13, 2023, from https://github.com/aws-greengrass/aws-greengrass-software-catalog

Greengrass Command Line Interface. (2023, 4). Retrieved 4 11, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/gg-cli.html

How AWS IoT Greengrass Works. (2023). Retrieved 10 3, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/how-it-works.html

How job configurations work. (2023, 4 25). Retrieved 4 25, 2023, from https://docs.aws.amazon.com/iot/latest/developerguide/jobs-configurations-details.html#job-rollout-abort-scheduling

Iftikhar, S., Gill, S. S., Song, C., Xu, M., Aslanpour, M. S., Toosi, A. N., . . . Uhlig, S. (2023). AI-based fog and edge computing: A systematic review, taxonomy and future directions. Retrieved 3 7, 2023, from Akamai Web site: https://www.akamai.com/our-thinking/cdn/what-is-a-cdn

Install AWS IoT Greengrass Core software with automatic resource provisioning. (2023, 3). Retrieved 3 27, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/quick-installation.html

Install the AWS IoT Greengrass Core software. (2023, 3). Retrieved 3 22, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/install-greengrass-core-v2.html

Interact with device shadows. (2023). Retrieved 3 20, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/interact-with-shadows.html

Local debug console. (2023, 4 11). Retrieved 4 11, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/local-debug-console-component.html

Manage data streams on Greengrass core devices. (2023). Retrieved 3 20, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/manage-data-streams.html

Meta-aws recipes. (2023). Retrieved 3 10, 2023, from https://github.com/aws4embeddedlinux/meta-aws/tree/master/recipes-iot

MicroPython. (2023, 4 13). Retrieved 4 13, 2023, from https://docs.micropython.org/en/latest/esp32/tutorial/intro.html

MQTT 3.1.1 broker (Moquette). (2023, 4 14). Retrieved 4 14, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/mqtt-broker-moquette-component.html

New persistent storage spooler(s). (2023, 4 17). Retrieved 4 17, 2023, from https://github.com/aws-greengrass/aws-greengrass-nucleus/issues/825

Premsankar, G., Di Francesco, M., & Taleb, T. (2018). Edge Computing for the Internet of Things: A Case Study. doi:https://doi.org/10.1109/JIOT.2018.2805263

Raspberry Pi 4 Tech Specs. (2023, 3). Retrieved 3 27, 2023, from https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

Sachin, K., Tiwari, P., & Zymbler, M. (2019). Internet of Things is a revolutionary approach for future technology enhancement: a review. *Journal of Big Data*. doi:https://doi.org/10.1186/s40537-019-0268-2

Setting up AWS IoT Greengrass core devices. (2023). Retrieved 3 10, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/setting-up.html

Shadow manager. (2023, 4 14). Retrieved 4 14, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/shadow-manager-component.html

Sharifi, L., Rameshan, N., Freitag, F., & Veiga, L. (2014). Energy Efficiency Dilemma: P2P-cloud vs. Datacenter. doi:10.1109/CloudCom.2014.137

Sodabathina, R., Shan, J., & Ulloa, M. (2022, 11). *Building event-driven architectures with IoT sensor data*. Retrieved 10 3, 2023, from https://aws.amazon.com/blogs/architecture/building-event-driven-architectures-with-iot-sensor-data/

umqtt.simple. (2023, 4 17). Retrieved 4 17, 2023, from https://mpython.readthedocs.io/en/master/library/mPython/umqtt.simple.html

Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core. (2023, 4 14). Retrieved 4 14, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/interprocess-communication.html

What is AWS IoT Analytics? (2023, 23 4). Retrieved 23 4, 2023, from https://docs.aws.amazon.com/iotanalytics/latest/userguide/welcome.html

What is AWS IoT Greengrass? (2023). Retrieved 2 23, 2023, from https://docs.aws.amazon.com/greengrass/v2/developerguide/what-is-iot-greengrass.html