Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in
Ingegneria Informatica

Secure Authentication Methods in Edge Computing Architectures using AWS and Open-Source Prototypes

Relatore:
Prof. Carlo Ferrari

Laureando:
Michele Libralato
1206604

ANNO ACCADEMICO 2023-2024

Data di laurea 05 Marzo 2024

Abstract

This document proposes an analysis of authentication methods in the context of edge computing architectures. As edge architectures become more widespread, it becomes increasingly important to create efficient infrastructures to enable the secure exchange of information between the cloud and IoT devices. Typically, edge nodes are used to record and preprocess information from IoT devices, in order to later share it with the cloud infrastructure. Despite the numerous efficiency advantages, implementing a stable and secure infrastructure poses significant challenges. In terms of security, one of the important points to explore is the management of authentication between different types of devices, seeking an authentication process that requires limited resources that can be executed on IoT devices and edge nodes. This study focuses on exploring authentication mechanisms and presents two prototypes: the first one based on AWS systems and the second based on an Open Source approach that uses Mosquitto to achieve authentication with the MQTT protocol.

Contents

List of Figures

Chapter 1

Introduction

Edge and Fog computing are two paradigms that have been rapidly evolving in recent years, enabling data processing near the source, reducing latency, and improving real-time decision-making capabilities. These infrastructures find applications in various domains, ranging from autonomous vehicles to Industrial Internet of Things (IIoT) systems.

However, after analyzing numerous academic articles on the development of Edge computing and Fog computing infrastructures, one recurring issue that needs further exploration is data security within these networks, particularly the development of a secure authentication system.

Based on the principles outlined in ISO/IEC 7498-2 concerning "Security Architecture," data transmitted between various devices must remain intact throughout the network. Access, modification, and reading of data should only be performed by authorized agents, through control mechanisms to restrict data access and sender authenticity verification.

As the complexity of the infrastructure increases, so do the vulnerabilities of the network and its corresponding security threats. The distributed and often resource-constrained nature of edge devices makes them susceptible to a range of threats, including unauthorized access, data breaches, and cyberattacks. It is imperative to ensure data security while conserving energy, maintaining low latency, and using limited device memory [43].

Several works propose complex authentication systems to guarantee correct access to data [31], through the use of the HTTPS protocol [19] to the Blockchain.

Specifically, this work delves into authentication methods applicable between the various components of an edge infrastructure. It presents two prototypes: one using Amazon Web Services (AWS) and another leveraging Mosquitto, an open-source message broker for the Internet of Things. These proposals serve as examples to demonstrate how to implement a simple yet secure authentication method within the realm of Edge and Fog computing.

Chapter 2

Fog Computing and Edge Computing

Fog computing is a type of distributed architecture that enables the seamless provision of resources (including data storage, computing power, etc.) to connect the Cloud with IoT devices. There isn't a precise definition of Fog computing, but the National Institute of Standards and Technology defined it in 2018 as "a horizontal, physical or virtual resource paradigm that resides between smart end-devices and traditional cloud computing or data center." [24].

From this definition, Fog computing can be seen as an extension of the Cloud paradigm aimed at improving latency, the available bandwidth for connections between the Cloud and IoT devices, and the Quality of Service (QoS) offered by applications developed within this infrastructure.

A key characteristic of Fog computing, being a horizontal architecture, is its need for a widespread geographical distribution of nodes to ensure almost immediate processing of data sent by edge nodes and subsequent analysis at the Cloud level.

The concept of Fog networking or fogging arises from the necessity to extend Cloud computing to accommodate the vast number of IoT devices connected to the Cloud and consequently manage the large volume of real-time data generated by these devices.

Edge computing, although somewhat similar, is different. In this case, it refers to a distributed system where some data processing or storage occurs directly at the edge node, which can be either the IoT device itself or a server very close to the edge node [23]. This requires the edge node to have significant computational capacity and resource availability to process data in real-time, which can then be used directly by the device itself and/or shared with the Cloud. In this scenario, there may not be an intermediate layer for load-balancing data processing; edge nodes are connected directly to Cloud servers and among themselves.

An example of this infrastructure is an autonomous vehicle. In this case, sensor data needs to be processed as quickly as possible to provide an immediate response and make real-time decisions. If the data is sent to the Cloud for processing, the latency could be too high, posing risks for driving decisions. However, the system can communicate with the Cloud to share statistical data [45].

Both Fog and Edge computing concepts can be applied in various domains and offer benefits such as reduced latency, reduced bandwidth usage, and less storage space consumption, ultimately improving service quality. However, they also present new challenges, including communication security among different nodes.

Security in these systems can be complex to address. There are multiple data exchanges between various nodes, not just between clients and servers. Therefore, to ensure secure communication it is necessary to consider numerous aspects, such as sharing intact data from verified sources.

Additionally, the speed of authentication and authorization between different levels (IoT nodes with Fog nodes or Fog nodes to the Cloud) must be taken into consideration to avoid creating a slow connection as well as guaranteeing a secure connection.

This study will delve into the aspect of authentication between different nodes, which can be edge or fog nodes, starting from a simple architecture that involves the exchange of publicly available information.

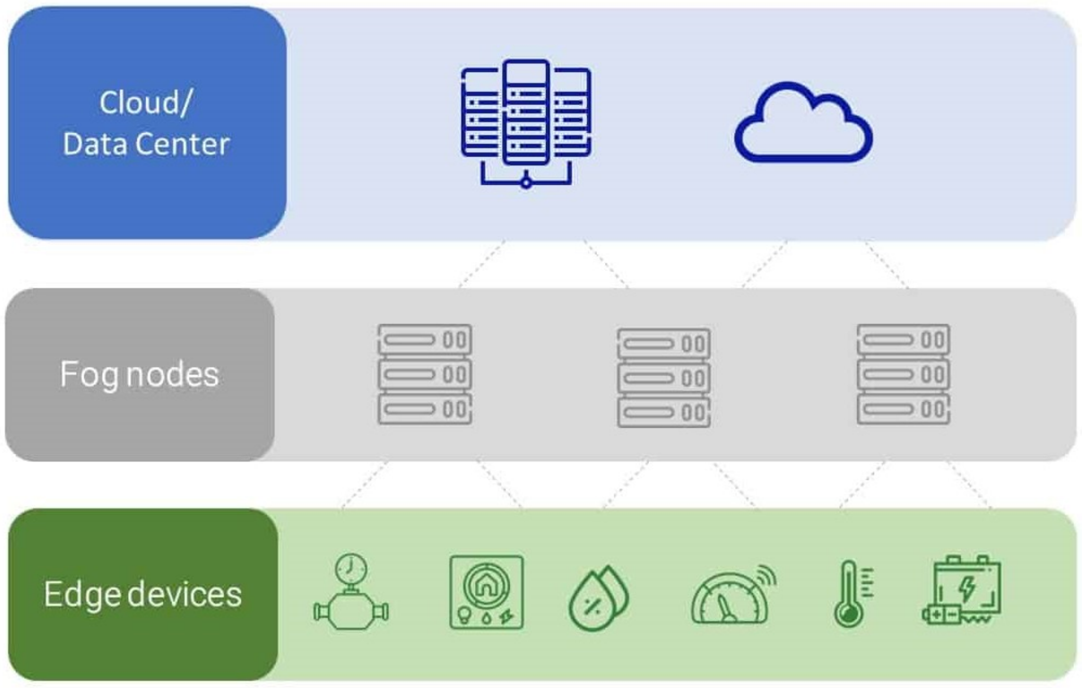


Figure 2.1: Fog and Edge computing infrastructure

Chapter 3

Authentication methods

3.1 Security Principles

Saltzer and Schroeder compiled a list of principles as mechanisms for securing operating systems. These principles remain relevant today as guidelines for Edge, Fog, and Cloud approaches. The original principles are as follows [32]:

- **Least Privilege**: Each user and/or process should employ the minimum privileges necessary to perform its task.
- **Separation of Privilege**: The security mechanism should satisfy more than one condition.
- **Least Common Mechanism**: Minimize the use of mechanisms shared by multiple users.
- **Economy of Mechanism**: Implement a mechanism that is simple, easy to test, and validate.
- **Complete Mediation**: Every access must be checked for authorization.
- **Fail-Safe Default**: Access rights should only be acquired through explicit request, and access decisions are based on permissions.
- **Open Design**: The security mechanism should not be secret. This allows for the use of encryption systems where algorithms are known but protection keys are kept secret.
- **User Acceptability**: The security mechanism should be user-friendly.

In addition, two additional principles are introduced:

- **Work Factor**: Implement robust security measures that are challenging to breach, discouraging hacking efforts.
- **Compromise Recording**: Record attacks to detect unauthorized usage.

In networks where nodes vary in nature, and communication is not solely between a user and a machine, these principles retain their validity. It is, therefore, crucial to establish a system that adheres to (most of) these principles.

3.2 Authentication Methods

It can be interesting to analyze methods commonly and widely used in other situations, such as User-Machine or Machine-Machine systems. The Edge or Fog architecture is made up of connections between different types of nodes, from IoT devices to Cloud microservices. From this analysis, we can understand which approach to use as an authentication method in an Edge or Fog computing infrastructure. In particular, we can examine client-server systems and communication between clients and Cloud microservices.

3.2.1 Client-Server

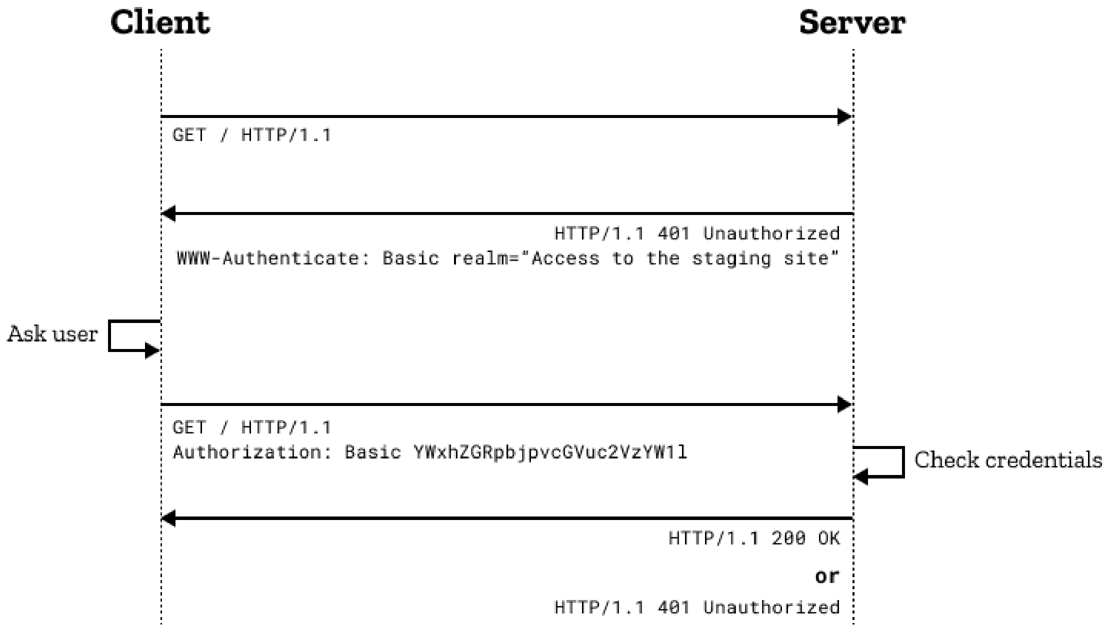
Regarding the classic client-server infrastructure, a practical example is the communication between a web application (client) and a server hosting the application's backend (with a database). In this scenario, the user logs in to use the application. The authentication, in this case, is typically referred to as monolith authentication. Monolith authentication refers to a traditional and centralized authentication system, where all user management and authentication mechanisms are handled by a single server [33].

While this system can centralize authentication functionalities (user registration, login, password management, etc.) in one place, making it easier to manage user data and policies, it also comes with limitations and challenges in terms of maintenance, scalability, and vulnerability to security risks [40].

The protocol commonly used in this context is the HTTP protocol [27] (defined by RFC 7235), which allows the client to send authentication information to the server. Here's a basic outline of the requests and responses between the client and server:

1. The client contacts the server and receives a 401 Unauthorized response from the server, requesting the client to send credentials.
2. The client sends the credentials in the message header after prompting the user to enter them.
3. Finally, the server responds with a 200 OK message if the credentials are correct, or a 401 Unauthorized message if they are not.

Figure 3.1 is a basic schema illustrating the requests and responses between the client and server.



```
GET / HTTP/1.1

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="Access to the staging site"

GET / HTTP/1.1
Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l

HTTP/1.1 200 OK
HTTP/1.1 401 Unauthorized
```

Figure 3.1: HTTP authentication scheme from MDN documentation [27]

3.2.2 Microservices

In a Cloud infrastructure composed of microservices, the authentication process differs because the system consists of various independent components (databases, serverless applications, storage) where a service must authenticate when communicating with another service within the same infrastructure or when the end user needs to query a cloud service [18].

Among the various approaches used in this case [49]:

Edge-level authentication

Through the use of an API Gateway. This centralizes authentication, allowing access control for each microservice. The API gateway provides the flexibility of a central interface for service utilization. If a service or user needs to access a microservice, it sends the request to the API gateway, which routes it to the requested service, as shown in figure 3.2.

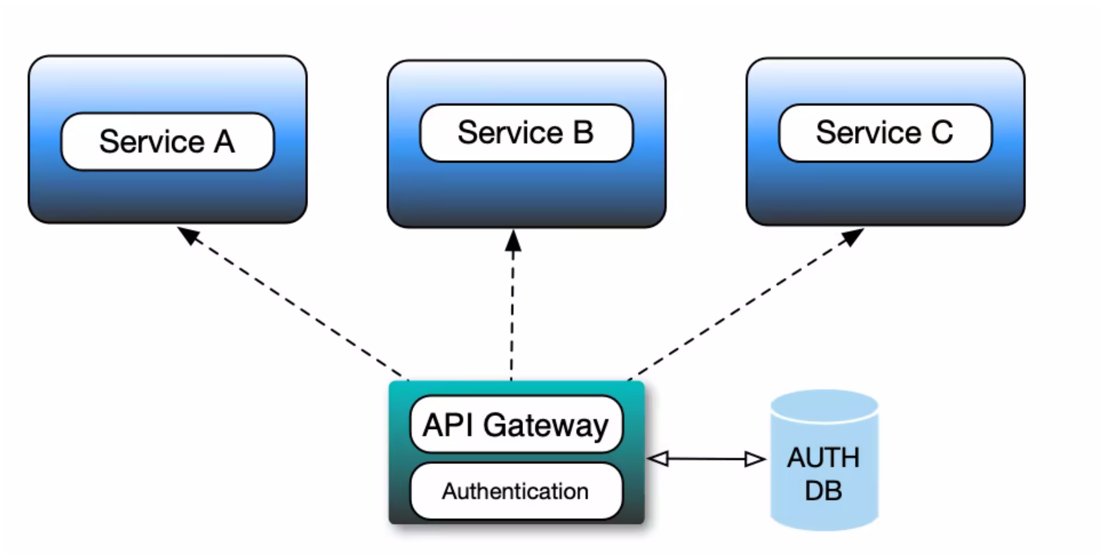


Figure 3.2: Implement Authentication Through the API Gateway [49]

Service-level authorization

Authentication and authorization are managed independently in each service. This approach allows each service to develop its own authentication, creating systems that can be highly independent of each other. However, this leads to duplicated code, the need to manage user data not owned by the service, and can be challenging to maintain (figure 3.3).

External entity identity propagation

A system that decides how to ensure authentication based on the user's context. A service is developed with the sole purpose of controlling network access, leading to a single point of failure and increased latency for authentication requests (figure 3.4).

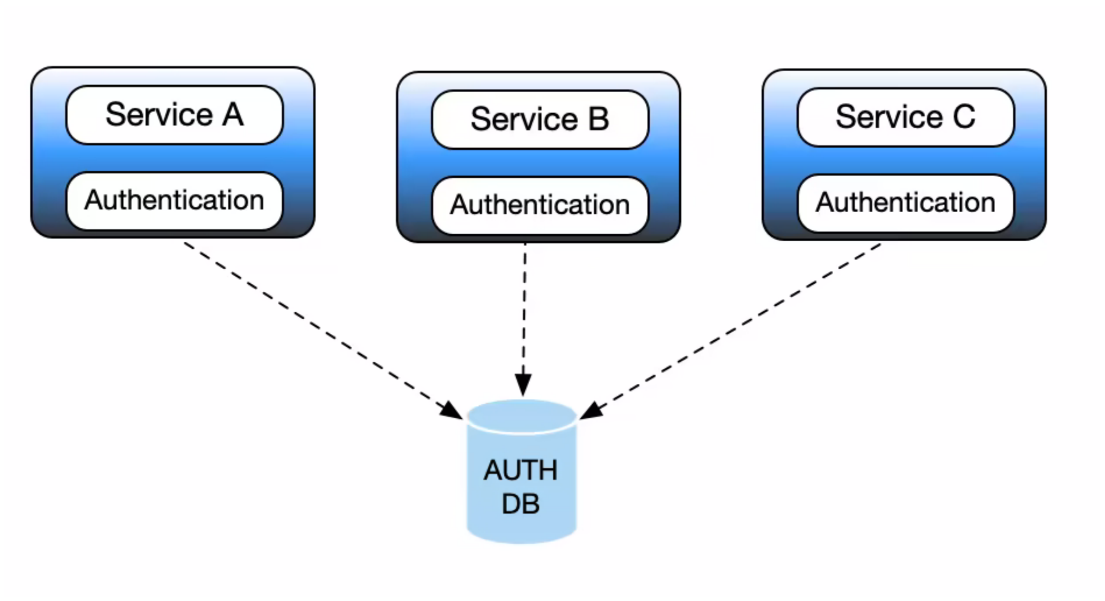


Figure 3.3: Implement Authentication on Each Microservice [49]

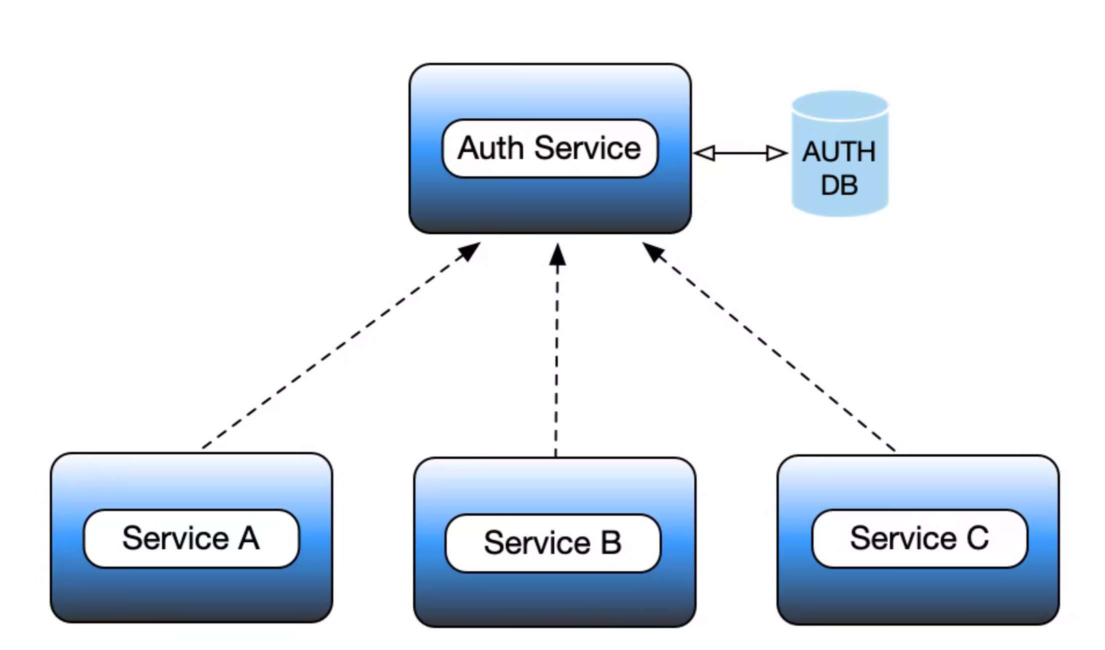


Figure 3.4: Implement Authentication Through an Authentication Service [49]

Also, in this case, communications between the parties generally take place using the HTTP protocol. There are various authentication mechanisms used to securely transmit information between the two parties. These include:

Username and Password

The basic standard where the user provides a personal ID and a secret code to access the resource.

Multi-factor Authentication (MFA)

Involves combining at least two user-provided factors for authentication:

- **Knowledge:** An element known by the user (username and password).
- **Possession:** An element the user possesses (one-time password OTP).
- **Biometrics:** An intrinsic element of the user (fingerprint).
- **Location:** The user's location.

Certificates

Digital documents used to identify a user or device and to associate that identity with a public key. Certificates are issued by a Certificate Authority (CA). The certificate issued by the CA binds a specific public key to the name of the entity the certificate identifies. Hence, only the public key certified by the certificate will work with the corresponding private key owned by the entity identified by the certificate.

Biometric

Involves authenticating using physical characteristics of the user (fingerprints, facial recognition).

JSON Web Token (JWT)

JWT is an open standard (defined by RFC 7519) consisting of a JSON-formatted signed token containing a set of claims that identify the sender's identity. The token comprises the header to describe the signature algorithm, the payload containing claim data, and the signature created by concatenating and encoding the header and payload in Base64 and then signing it with a private key using the algorithm specified in the header. Figure 3.5 shows the request-response schema using JWT tokens.

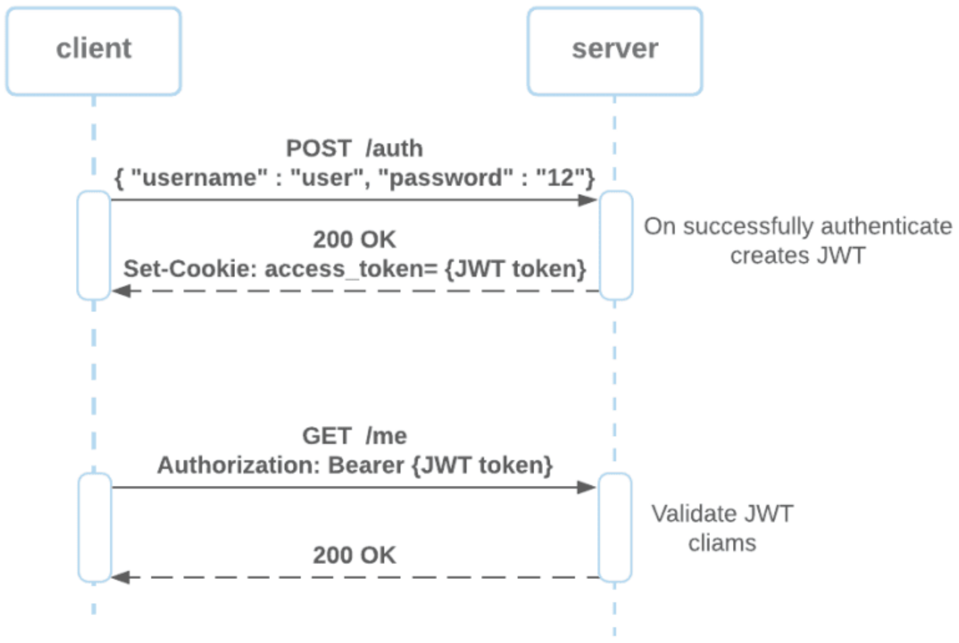

client
server
POST /auth
{ "username" : "user", "password" : "12"}
200 OK
Set-Cookie: access_token= {JWT token}
On successfully authenticate
creates JWT
GET /me
Authorization: Bearer {JWT token}
200 OK
Validate JWT
cliams

Figure 3.5: Request-response schema using JWT tokens [44]

3.3 Example for Secure Authentication - Blockchain

Several articles have been published to analyze and find a solution to ensure the security and privacy of data in Intelligent Transportation Systems (ITS) using blockchain technology. ITS systems can leverage edge or fog architectures based on where the data is processed. A specific example is presented in the article "Blockchain-Based Privacy-Preserving Authentication Model Intelligent Transportation Systems" [30] which elaborates on a model to ensure user privacy and communication security in the transportation network.

In this case, the blockchain, based on a distributed database, creates blocks that record data transactions, timestamps, and the hash value of the previously linked block. Using the blockchain system in ITS, allows the elimination of a single security authorization center in the network. For vehicle authentication, the public key is used, and applications like Ethereum provide a transparent and secure data storage environment.

In a generic ITS, a transaction block is created when there is a new ITS transaction request. The created blocks are broadcasted to all Vehicle Nodes and validated. Upon validation, the blocks are added to the chain to conclude the transaction. See the transaction diagram in Figure 3.6.

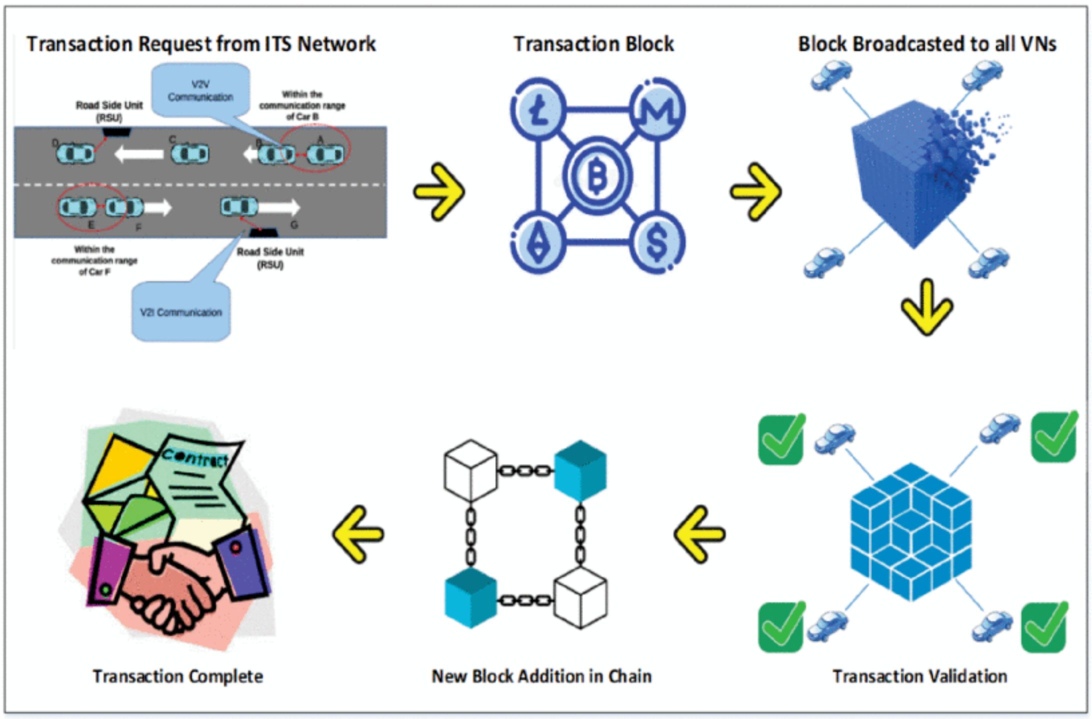

Transaction Request from ITS Network
V2V Communication
Road Side Unit (RSU)
Within the communication range of Car B
D
C
B
A
E
F
G
Within the communication range of Car F
Road Side Unit (RSU)
V2I Communication
Transaction Block
Block Broadcasted to all VNs
Transaction Complete
New Block Addition in Chain
Transaction Validation

Figure 3.6: Steps to add a transition request to a blockchain in ITS

This article explains the BPPAU (Blockchain-based Privacy-Preserving Authentication) model, ensuring user privacy and security using blockchain technology. It employs mechanisms for data storage, access, and processing via a smartcontract system, access control policy and on demand based functions. The blockchain network provides a peer-to-peer platform to support decentralized applications in ITS networks, including executing smart contracts and storage on Ethereum. The Transaction Phase is illustrated in the figure 3.7, including steps such as smart contract publication by vehicle nodes, data storage, updating the data identifier for the blockchain network, data access request, validation process by retrieving information from Trusted Execution, Service Provider, and Distributor Storage.

In these blockchain models, it is essential to consider the high processing requirements and the overall costs incurred for processing. Moreover, it emphasizes that these systems might not be very efficient in other types of networks where fast information processing is required while ensuring network security. This highlights that while blockchain utilization as an authentication system can be highly secure, it may not be suitable for all types of architectures.

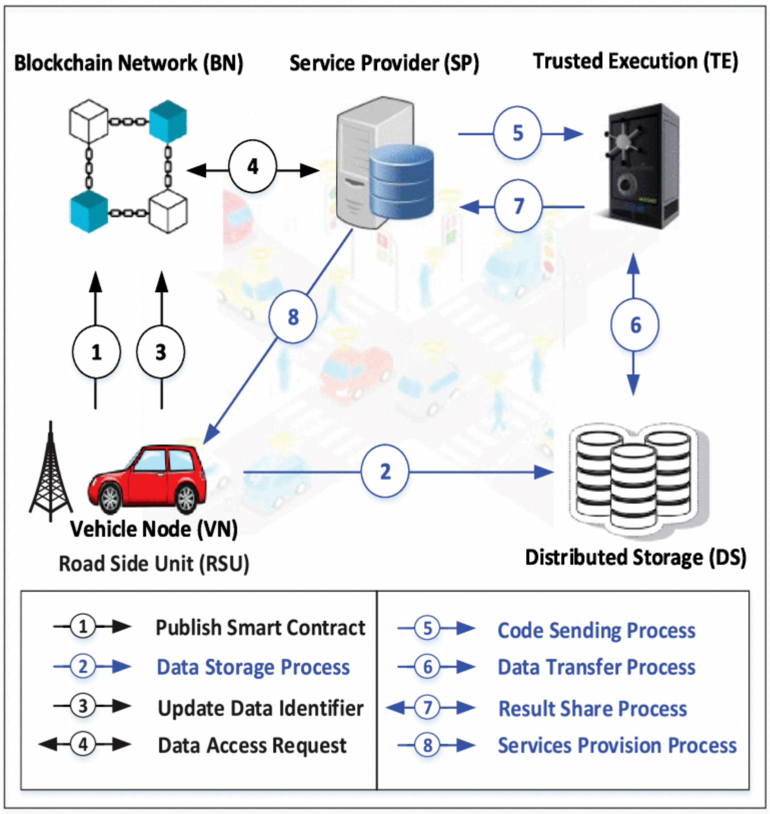


Figure 3.7: BPPAU system model from the article "Blockchain-Based Privacy-Preserving Authentication Model Intelligent Transportation Systems" [30]

3.4 Authentication between IoT devices

In the majority of Edge and Fog computing infrastructures, a substantial set of nodes is represented by IoT devices. Unlike nodes at the Fog and Cloud levels, authentication mechanisms between devices at the Edge level and other nodes can vary. In these networks, it is essential to decide on both the type of protocol to employ [1] for data transmission and the authentication method to be applied [47].

3.4.1 IoT Protocols

IoT protocols are standards that enable the exchange and sharing of data across the Internet among various devices. These protocols can be categorized into network protocols and data protocols.

Network protocols are responsible for connecting IoT devices and correspond to the Physical and Data Link layers of the ISO/OSI model. Common examples include Wi-Fi, LTE CAT 1, LTE CAT M1, NB-IoT, Bluetooth, ZigBee, and LoRaWAN.

On the other hand, data protocols, which operate at the Presentation and Application layers of the ISO/OSI model, are crucial for formatting data in a way that is usable by the end application. The choice of data protocol is fundamental when designing a system for a particular infrastructure, taking into account available resources and the computational capabilities of individual devices. Among the various data protocols, the most common ones include:

- AMQP
- MQTT
- HTTP
- CoAP
- DDS
- LwM2M

In this analysis, we delve deeper into the MQTT protocol, which, unlike the HTTP protocol, is the most commonly used for communication among edge devices.

3.4.2 MQTT

MQTT, which stands for Message Queuing Telemetry Transport, is a lightweight publish-subscribe messaging protocol situated at the application layer of the ISO/OSI model [25]. It was primarily developed for M2M telemetry, catering to small and low-power devices, and has found widespread use in IoT devices.

The MQTT protocol comprises two main components: the client and the broker. Communication originates from the client, which can function as either a publisher or a subscriber, initiating the interaction with the MQTT broker. The broker acts as an intermediary between the clients, managing communication and efficiently routing messages between publishers and subscribers. It takes responsibility for client connection management, authentication, and authorization.

Messages in MQTT are organized into topics, which the protocol uses for routing messages. The publisher can send messages to specific topics, while subscribers can subscribe to a topic to receive messages sent to it. The broker receives the messages and routes them to the clients subscribed to the respective topic. The figure below 3.8 illustrates the steps in communication using this protocol.

Some advantages of MQTT include [39]:

- Providing flexible and efficient communication.
- Enabling bidirectional communication.
- Scalability to support millions of devices.
- Implementation of secure communications using TLS (Transport Layer Security).
- Usability in less stable networks.

Regarding security, MQTT brokers offer various authentication methods, allowing clients to choose how they authenticate with the broker [29]. The selection of authentication methods should take into account the capabilities of both the broker and the client. Common authentication methods include Client ID, Username and Password, and X.509 Client Certificates.

In addition to authentication, MQTT data security can be protected through TLS or payload encryption. TLS security is a part of the most widely used TCP/IP

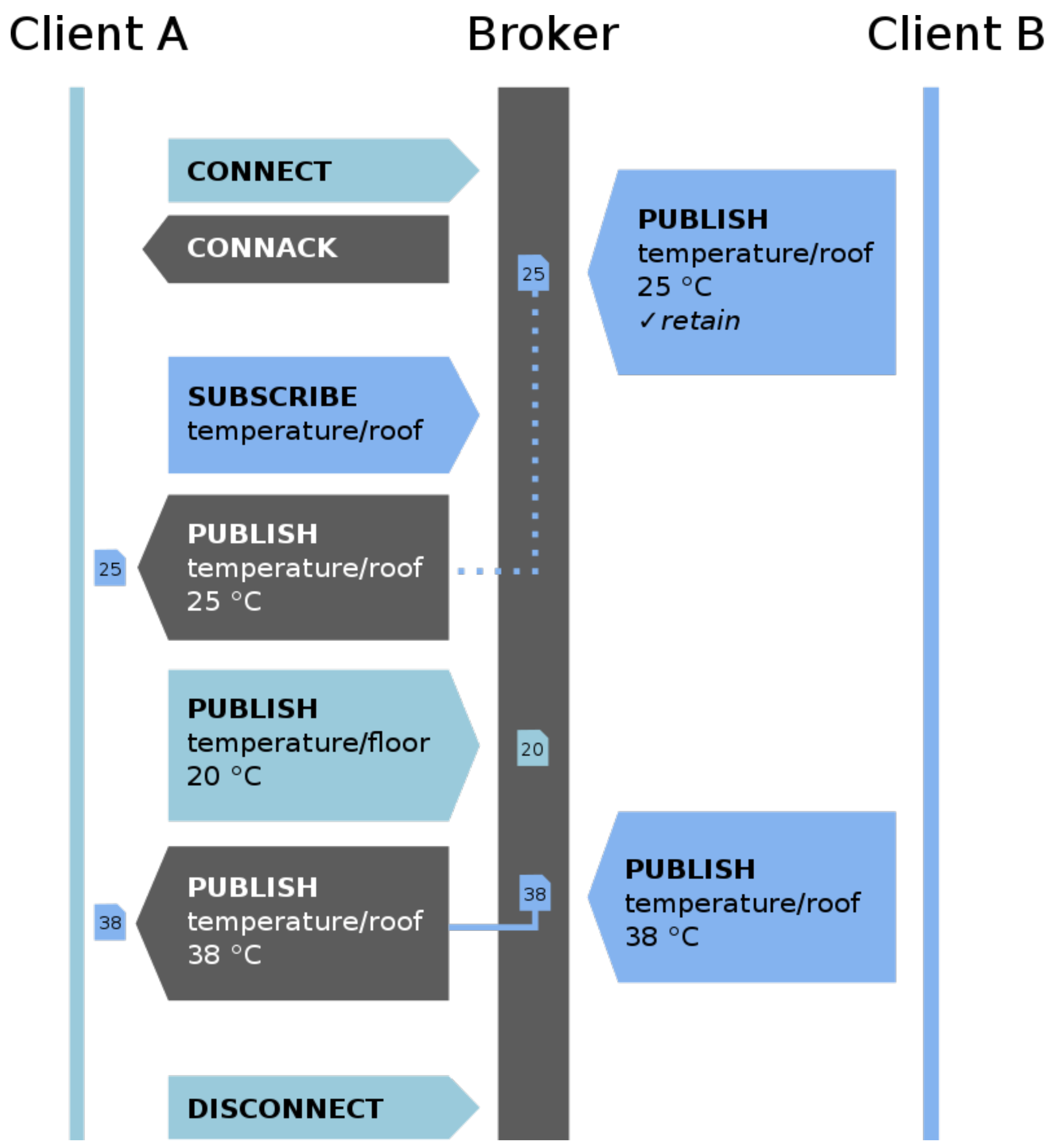


Figure 3.8: Example of communication via MQTT protocol

protocol and mainly creates an encrypted channel through which MQTT messages can be transmitted. TLS security requires the use of certificates.

One of the most popular and widely used open-source MQTT brokers is the Mosquitto broker, which will also be utilized in the prototype.

3.4.3 IoT Device Authentication

The issue of authentication in IoT devices revolves around establishing a trust model for secure communication between these devices and other nodes. Its primary purpose is to prevent unauthorized access to data. There are mainly three authentication methods:

1. **Password:** The device uses a password for authentication with another device.
2. **Symmetric Keys:** The two devices, or the device and the cloud, require the establishment of a shared symmetric key.
3. **Certificates:** A digital certificate is employed to identify the IoT device. The certificate is a signed data structure that associates the identity of the IoT device with a public key. It relies on asymmetric encryption to create a public-private key pair for the device.

Among these methods, certificates, and specifically X.509 certificates, are the most widely used and secure for authentication.

3.4.4 X.509 Certificates

To establish secure communication between an IoT device and an edge node and to identify the IoT device (IoT Identity Management [20]), an X.509 certificate will be used.

The X.509 certificate is an International Telecommunication Union (ITU) standard described in RFC 5280 [28] and is a digital document used to represent a device, user, or service. This document is primarily used in the context of a Public Key Infrastructure (PKI).

The digital document contains, in addition to the name of the entity and other identifying data, the public key, used to encrypt the certificate together with the corresponding private key, which must be kept secret. The Certificate Authority (CA) is responsible for issuing the certificate and signing it using the private key to ensure its authenticity.

Below is an overview of the structure of an X.509 v3 certificate [48]:

- Version Number
- Serial Number
- Signature Algorithm ID
- Issuer Name

- Validity Period
  - Not Before
  - Not After
- Subject Name
- Subject Public Key Info
  - Public Key Algorithm
  - Subject Public Key
- Issuer Unique Identifier (optional)
- Subject Unique Identifier (optional)
- Extensions (optional)
- Certificate Signature Algorithm
- Certificate Signature

In the prototypes described in this work the certificates will have the .pem format, therefore formatted in Base64 enclosed between "—–BEGIN CERTIFICATE—–" and "—–END CERTIFICATE—–". The SHA-256 algorithm is commonly used for encrypting the certificate.

Chapter 4

Prototypes

4.1 Introduction to Prototypes

Considering a general scenario where the edge infrastructure is relatively simple and consists of IoT devices as peripherals with limited computational capabilities, one can contemplate constructing a simple and easy-to-implement authentication system.

Practical examples could include weather parameter sensors installed in different cities that share collected data with a potential intermediate node and subsequently with the cloud. Another possible infrastructure is the simulation of a network of traffic surveillance cameras. The objective is to establish communication between a camera acting as an IoT device and a fog node responsible for collecting the data and transmitting it to a cloud database (such as an AWS database). In these examples, reference is made to cases where the processed data is not personal but in the public domain.

When working with IoT devices and edge nodes, using the MQTT protocol for data sharing is the most convenient and simple solution to implement. To understand how an authentication process using the MQTT protocol can function, two simple prototypes were developed. Initially, the AWS environment was utilized, which offers services for managing IoT and Edge infrastructures. Contemplating if it was possible to implement the same infrastructure using open-source frameworks, the second prototype uses NodeJS, Mosquitto an open-source MQTT broker, and Redis a non-relational database. The primary aim of these prototypes is to comprehend how authentication is managed within this type of infrastructure and to identify potential improvements compared to the current state-of-the-art practices.

4.2 Prototype using Amazon Web Services (AWS)

Figure 4.1: AWS Logo

Amazon Web Services (AWS) is a company that provides a wide range of cloud computing services and is the most globally widespread Cloud Service Provider [46]. For this reason, an initial solution was developed using several AWS services. To create a Fog architecture, it's essential to interconnect IoT devices, Fog nodes, and the Cloud. The AWS IoT Core service serves as the first step to create and manage IoT devices, while AWS Greengrass is used for the Fog nodes. In the prototype, these devices are simulated through EC2 instances, but a local PC or a Raspberry Pi can also be employed. Subsequently, the data generated and managed by the Edge and Fog nodes are stored in a table within a NoSQL database called DynamoDB, a Cloud service provided by AWS. The graph 4.2 represents the services used for this prototype.

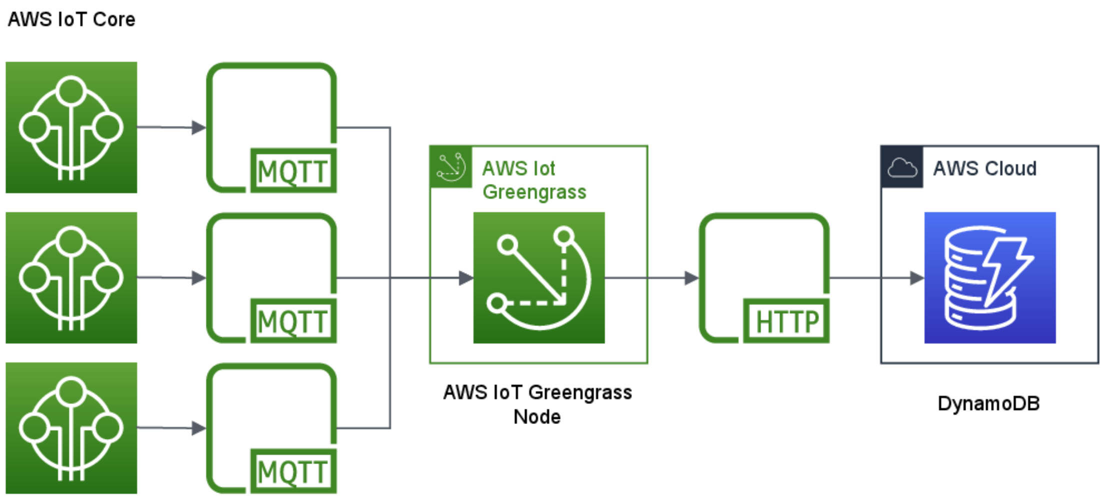


Figure 4.2: Example of fog infrastructure using AWS services

4.2.1 AWS IoT Core

AWS provides services for managing an Edge infrastructure, with AWS IoT Core being the primary service for this purpose. It allows for the creation and management of IoT devices, establishing connections between devices and other cloud services [14], figure 4.3.

Among the various supported communication technologies for devices, the MQTT protocol is included. Devices can be configured using the AWS SDK to create applications and send/receive messages. The entire configuration process can also be managed through the GUI in the AWS console.

Regarding authentication, the identity of IoT devices is verified through X.509 certificates. Generally, these certificates are created during the device registration process with AWS IoT Core, and they can be signed either by AWS IoT itself or by a Certificate Authority (CA) registered with AWS IoT. This approach allows for the management of certificate rotation and replacement when they approach expiration. Edge devices use the same certificate to find, connect, and authenticate to a Greengrass node via AWS IoT Greengrass cloud discovery. The certificate is then used to connect to both the AWS IoT Core service and the Greengrass device [2].

From here on, the IoT device created using AWS IoT Core can be referred to as a “client”.

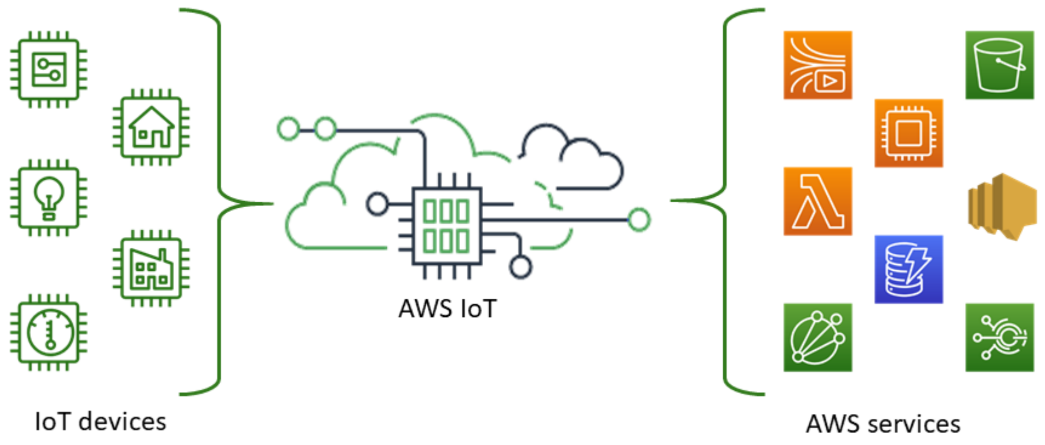


Figure 4.3: AWS IoT Core overview

4.2.2 AWS IoT Greengrass

An essential service for setting up a fog node is AWS Greengrass, which enables the configuration of an intermediate device to facilitate communication between client devices and other cloud services [13], as shown in figure 4.4.

In an edge device configured with Greengrass, there is the capability to develop software on the device to manipulate or analyze the generated or received data. Greengrass devices can securely communicate with AWS IoT Core clients and the Cloud. You can run applications called Components to manage various forms of communication between client devices and Cloud services. The Greengrass device can henceforth be referred to as "core".

For authentication, similar to the clients, X.509 certificates are employed. As demonstrated in the prototype, these certificates serve core devices to connect with client devices in AWS IoT Core and to upload components and configurations to the core device [5]. The core device establishes and communicates with the client device using the MQTT protocol [10].

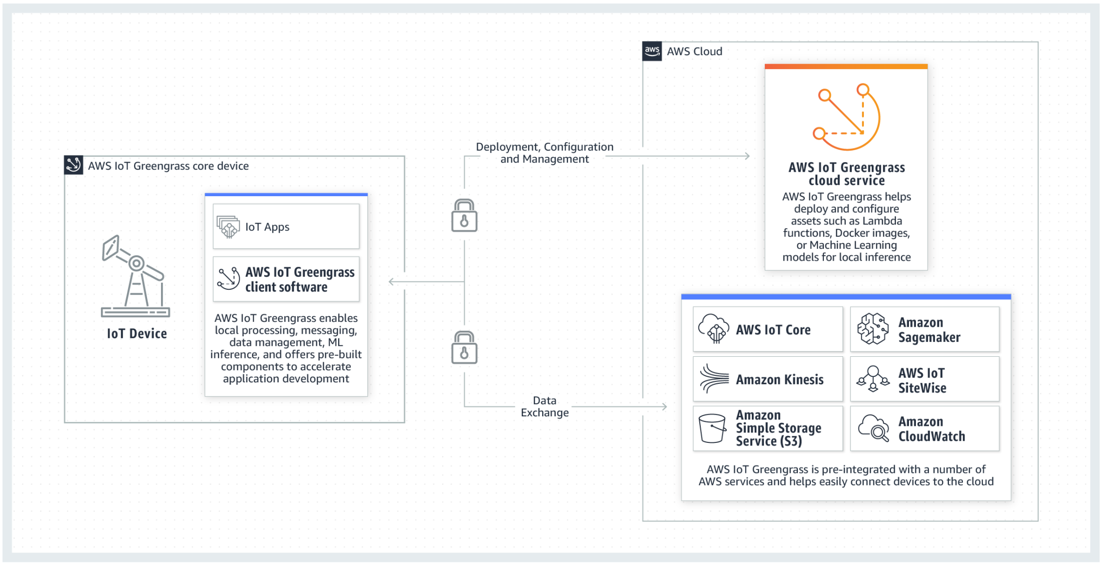

AWS Cloud
AWS IoT Greengrass core device
IoT Device
IoT Apps
AWS IoT Greengrass client software
AWS IoT Greengrass enables local processing, messaging, data management, ML inference, and offers pre-built components to accelerate application development
Deployment, Configuration and Management
Data Exchange
AWS IoT Greengrass cloud service
AWS IoT Greengrass helps deploy and configure assets such as Lambda functions, Docker images, or Machine Learning models for local inference
AWS IoT Core
Amazon Sagemaker
Amazon Kinesis
AWS IoT SiteWise
Amazon Simple Storage Service (S3)
Amazon CloudWatch
AWS IoT Greengrass is pre-integrated with a number of AWS services and helps easily connect devices to the cloud

Figure 4.4: AWS Greengrass sample connection overview

4.2.3 Amazon Dynamo DB

DynamoDB is used to save data in the Cloud, which is a fully managed NoSQL or non-relational database service provided by AWS, known for its fast performance and auto-scalability. Even though the example provided may not involve substantial data, DynamoDB can be used to create tables for storing and retrieving any volume of data, with automatic resource scaling based on traffic. Being a database tailored to handle vast data streams, all data is stored on SSDs and automatically replicated across multiple Availability Zones within an AWS region, ensuring immediate availability and data durability [12]. Access to the DynamoDB table occurs using the AWS SDK and authenticating through the use of the Secret Access Key and the Access Key connected to a specific user.

4.2.4 AWS Prototype Development

The creation and simulation of this prototype was done mainly following the official AWS documentation. Below are the main steps to simulate a generic infrastructure:

1. **Creation of virtual devices**
   Two Linux instances were created in EC2 to simulate an IoT device (referred to as the client device) and a Greengrass device (referred to as the core device). Both instances were t2.micro with 1 GB RAM and 8 GB storage.

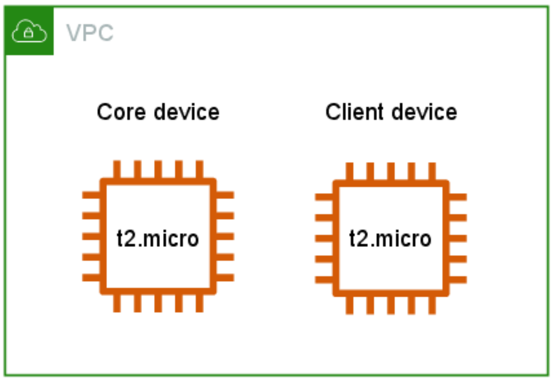


Figure 4.5: EC2 Instances

2. **Configuration of the client device** [4]
   After installing and updating Git, Python, and AWS CLI 2, the virtual device can be registered in AWS IoT using AWS CLI:

   ```
   aws iot create-thing --thing-name "device1"
   ```

   Device registration is necessary to create an endpoint representing the device's internet-connected address:

`a3luyee6pcu06d-ats.iot.eu-west-1.amazonaws.com`
After registering the client device, the necessary certificate for authentication can be created. This involves:

- Downloading a copy of the Amazon Certificate Authority (CA) certificate to identify the virtual device
- Generating a private key, public key, and X.509 certificate using AWS CLI
- Associating the newly created certificate with the IoT object in AWS IoT

To set up permissions, a policy can be created listing the actions the device can perform. The policy file can be associated with the registered device using AWS CLI. Finally, the necessary SDK can be installed to use it in an application. In this case, a sample application *basic_discovery.py* (see appendix 1.3) can be launched to publish an MQTT message to a specific topic, which is present in the Python SDK. The Python SDK for AWS IoT devices can be installed as follows:

```
git clone \
    https://github.com/aws/aws-iot-device-sdk-python-v2.git
```

The previously created parameters will be used to launch the sample code:

- CA certificate
- Device's X.509 certificate
- Private key
- Device endpoint

Client device components
Device
Certificate X.509
Policy
Python SDK

Figure 4.6: Client device components

3. **Configuration of the core device** [7]

   To configure the core device, it's necessary to install the AWS IoT Greengrass Core software. This can be done via the AWS console by following the steps for device configuration. [9]

   After installing or updating Java (the environment required by the AWS IoT Greengrass Core software) and configuring the AWS credentials to download the necessary AWS resources (Access Key and Secret Access Key), it is possible to download the installer using the following command:

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/\
       greengrass-nucleus-latest.zip > \
       greengrass-nucleus-latest.zip && \
       unzip greengrass-nucleus-latest.zip -d GreengrassInstaller
   ```

   Then, the installer can be run with Java:

   ```
   sudo -E java -Droot="/greengrass/v2" \
       -Dlog.store=FILE \
       -jar ./GreengrassInstaller/lib/Greengrass.jar \
       --aws-region eu-west-1 \
       --thing-name GreengrassQuickStartCore-18b9c6e9bd5 \
       --thing-group-name GreengrassQuickStartGroup \
       --component-default-user ggc_user:ggc_group \
       --provision true \
       --setup-system-service true \
       --deploy-dev-tools true
   ```

   This package allows you to:

   - Install and configure the Greengrass Core software as a system service (Nucleus).
   - Set up the Greengrass CLI component, a command-line tool for developing Greengrass components.
   - Configure the user created earlier (`ggc_user`) as a system user for running applications.
   - Connect the device to AWS IoT and the group where all edge network devices will be grouped (`GreengrassQuickStartGroup`).

- Download the X.509 certificate and default permissions to identify the core device from the client device during authentication and connection via Cloud Discovery.

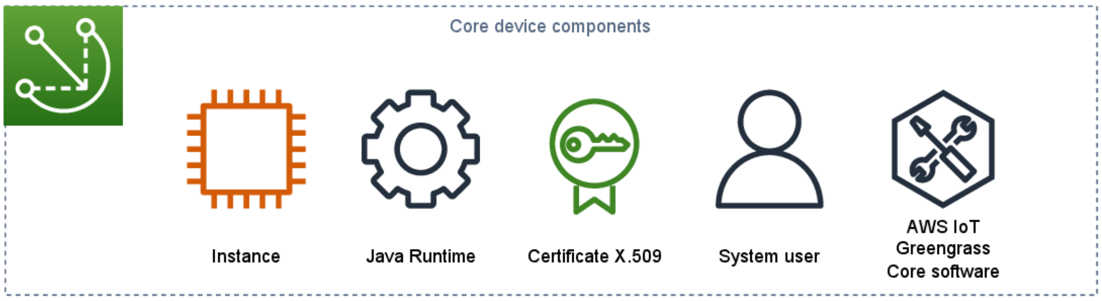


Figure 4.7: Core device components

4. **Creation of a component in the core device** [8]
   Using the Greengrass CLI, components in the core device can be managed. A component is a software module executed within the device. It allows the creation and management of application blocks for use within the instance or other edge nodes. A component consists of:

   - Recipes: a metadata file with parameters for executing the code
   - Artifacts: files containing the code to be executed (scripts, compiled code, etc.)

   A new component is created in the core device, including:

   - The recipe file `com.example.clientdevices.MyHelloWorldSubscriber-1.1.0.json` (see appendix 1.1)
   - The artifact `hello_world_subscriber.py` (see appendix 1.2)

   The artifact `hello_world_subscriber.py` of the core device uses the IPC service to subscribe to the `clients/device1/hello/world` topic and read messages that it receives. The AWS IoT Greengrass Core interprocess communication (IPC) library in the AWS IoT Device SDK (in this case, AWS IoT SDK for Python) is useful for communicating with the AWS IoT Greengrass nucleus and other Greengrass components. Additionally, besides reading messages sent to a specific topic, the component saves the read messages in DynamoDB, the AWS Cloud service for managing non-relational databases. Using the Greengrass CLI, the component can be deployed in AWS IoT Greengrass:

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \
    --recipeDir ~/greengrass-2-pubsub/recipes \
    --artifactDir ~/greengrass-2-pubsub/artifacts \
    --merge "com.example.clientdevices.MyHelloWorldSubscriber=1.1.0"
```

By deploying the component, it becomes active in the core device.

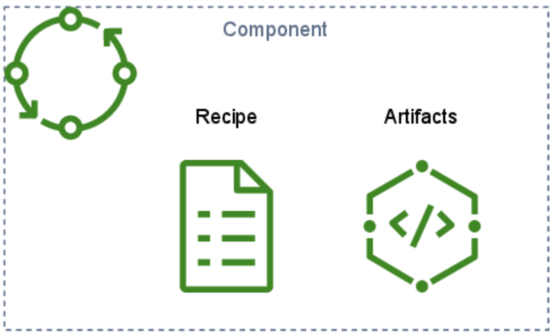


Figure 4.8: Component structure

5. **Communication between devices** [11]
   As a final step, communication between the core device and the client device can be established via MQTT. In this specific scenario, the configuration of AWS IoT Cloud Discovery (a service within the Greengrass group) can be implemented, allowing the client device to search and connect to the core device by sending a request to the AWS IoT Greengrass Cloud service to find the recipient core device. This service shares the IP and certificates that the client can use for the connection.
   An additional step to do before activating communication between devices is to upload some components on the core device to use Cloud Discovery and associate the client device with the core device:

   - `aws.greengrass.clientdevices.Auth`: used for authenticating and authorizing the client device
   - `aws.greengrass.clientdevices.mqtt.Moquette`: represents the MQTT Moquette broker
   - `aws.greengrass.clientdevices.mqtt.Bridge`: used for relaying messages from the client device to AWS IoT Core and for reading messages from the console

- `aws.greengrass.clientdevices.IPDetector`: shares the MQTT broker's endpoints with AWS IoT Greengrass.

The Core Discovery configuration can be done through the AWS console, setting policies for the components `aws.greengrass.clientdevices.Auth` and `aws.greengrass.clientdevices.mqtt.Bridge`.
Through the AWS console you can therefore associate the client and the core device. This allows the client device to use cloud discovery to obtain connection information and the certificates of the core device.
At this point, still on the core device, the component can be created to communicate via MQTT with the client device. The component's MQTT bridge configuration is set to read messages received on a specific topic (e.g., `clients/+/hello/world`).
Deploying the previously created component with the AWS Greengrass CLI makes it active, and the component logs can be checked using the following command:

```
sudo tail -f /greengrass/v2/logs \
    /com.example.clientdevices.MyHelloWorldSubscriber.log
```

Now using the client device, with the Python SDK previously installed, the `basic_discovery.py` (see appendix 1.3) sample code can be executed on the client device to initiate Cloud Discovery:

```
python3 basic_discovery.py \
    --thing_name device1 \
    --topic 'clients/device1/hello/world' \
    --message 'Hello World!' \
    --ca_file ~/certs/Amazon-root-CA-1.pem \
    --cert ~/certs/device.pem.crt \
    --key ~/certs/private.pem.key \
    --region eu-west-1 \
    --verbosity Warn \
```

This way, the client device performs Cloud Discovery and sends messages to the `clients/device1/hello/world` topic via MQTT, which are then read by the core device.

4.2.5 Comments and Considerations

The client device, when executing the script `basic_discovery.py`, uses the `mqtt_connection_builder` method, which allows it to manage the session duration. Among the parameters of this method defined in the AWS SDK, there is `keep_alive_secs`, which establishes the maximum period of inactivity in seconds (1.5 x seconds) after which the connection is terminated. However, the session in AWS IoT Core cannot have a duration of more than 30 days.

Exploring authentication between the various components using the MQTT protocol, all devices use an X.509 certificate [15] and cryptographic keys. The certificates that ensure the authenticity of the device's identity are issued by the AWS IoT Certificate Authority (CA), in this particular case, the Amazon Trust Services (ATS) root CA certificate.
The certificates and public keys, besides authenticating a device, are also used to establish a Transport Layer Security (TLS) connection. The certificates are created and stored in the Cloud during device configuration. Like all certificates, these certificates have a validity period, meaning they expire after a certain amount of time.
In AWS, there is a certificate rotation method to be performed before they expire. In AWS Greengrass, the core device certificate expires every 5 years. Additionally, during the connection to the MQTT broker, a local server certificate is created, which can last for 7 days (a parameter that can be changed). This certificate is used by the client device for mutual authentication. The limited duration serves to prevent attacks that can steal the certificate and private key to impersonate the core device. The rotation occurs automatically 24 hours before the expiration, during which all devices connected to the MQTT broker are temporarily disconnected, and the broker is restarted to allow client devices to reconnect [6].

Data updating with the DynamoDB service is done using the Python SDK (`boto3`). In this case, authentication is performed using AWS credentials: the Access Key and Secret Access Key created through the IAM (Identity and Access Management) service by creating a dedicated user in AWS [3]. The connection between AWS services and the device occurs through HTTPS. The SDK creates an API POST request to the AWS service using the set keys for authentication.

Analyzing the connection management in AWS, it can be deduced that several conditions are considered for authentication refresh:

- After a certain period of inactivity, through the `keep_alive` parameter
- As a best practice, periodic authentication refresh after a long connection period
- Certificate expiration and automatic renewal
- Updates to the IoT device or the edge node requiring disconnection
- Connection interruption due to various network errors

The authentication process employing X.509 certificates and AWS credentials (Access Key and Secret Access Key) enables secure connections for various devices across different levels, ranging from the edge to the Cloud.

4.3 Open-Source Prototype

To have an open-source alternative to using AWS services, it is possible to develop a prototype using the Node.js module MQTT.js [17], the open-source MQTT broker Mosquitto [34] and Redis [41]. With a Node.js script, it is possible to simulate communication between two devices. In this scenario, there is a publisher script representing a weather station that sends parameters to the broker to share the current wind situation (see appendix 1.4). A subscriber script receives the message and stores it in Redis, a NoSQL database (see appendix 1.5). Therefore, the subscriber acts as an intermediate node between the edge device (publisher) and the Cloud. In the subsequent paragraphs, the two devices will simply be referred to as the publisher and the subscriber.
For connecting to the Mosquitto brokers, both Node.js scripts can accept an argument to set the authentication options to use:

- option_1: authentication to the test Mosquitto broker using username and password.
- option_2: authentication to the test Mosquitto broker using X.509 certificates.
- option_3: authentication to the Mosquitto broker running in a Docker container using username and password
- option_4: authentication to the Mosquitto broker running in a Docker container using X.509 certificates.

Figure 4.9: Open-source prototype components

4.3.1 Redis

Redis (short for REmote DIctionary Server) is an open-source database where various key-value data structures can be stored in memory rapidly [41]. Its use cases vary from caching, session management, pub/sub services, to leaderboards. In this prototype, the Redis database runs within a Docker container (see appendix 1.8). The redis service is started by setting a password through the Redis configuration that must be used when connecting the devices (redis-password). This Redis Docker container represents a Cloud database service, thus forming the Cloud layer in the edge architecture.
Within the Node.js script, which simulates the intermediate device, the connection to the Redis server occurs through the npm package 'redis' [42]. Specifically, the subscriber connects to the Redis server via the default port 6379, passing the REDIS_PASSWORD as a parameter, stored as an environment variable in the device.

4.3.2 Mosquitto Test Server

For the initial test, both devices connect to the test server provided by Mosquitto: `test.mosquitto.org`. The publisher sends a message containing wind parameters under the topic 'wind'. The subscriber subscribes to a topic (in this case, 'wind') and listens for incoming messages. Once the broker shares and forwards the message to the subscriber, the device prints the published message under the 'wind' topic and connects to the Redis container to store it in the database.
To establish an authenticated and secure connection with the test server, ports

8885 and 8884 can be used [38]. In the Node.js script, an MQTT client is initialized using the npm `async-mqtt` package [17], where the methods are the same as in MQTT.js [**71.1**], except that the functions return promises instead of accepting callbacks. Both of these ports create an encrypted connection. To establish it, the `mosquitto.org.crt` certificate authority file can be downloaded to verify the server's connection.

With port 8885, an encrypted and authenticated MQTT connection can be created using a username and password. To test a connection, `rw` can be used as the username and `readwrite` as the password. This allows writing and reading messages on a topic. Below are the options used to connect with port 8885:

```
const options = {
    port: 8885,                 // Port for encrypted connection
    host: 'test.mosquitto.org', // Mosquitto broker URL
    clientId: '1234',           // Unique client ID
    username: 'rw',             // MQTT broker username
    password: 'readwrite',      // MQTT broker password
    protocol: 'mqtts',          // Encrypted connection protocol
    ca: [ca],                   // Root CA cert. for encryption
};
```

With port 8884, an encrypted and certified MQTT connection can be created using the client's certificate. To create the certificate, OpenSSL can be used from the command line [35]. Here are the steps for generating the private key and the certificate signing request (CSR):

```
openssl genrsa -out client.key  // Generating a private key
openssl req -out client.csr -key client.key -new  // Generating a CSR
```

Once the CSR has been generated, it is possible to send the content to the test server via the `https://test.mosquitto.org/ssl/` page and receive the content of the certificate to be used in connection requests.
Below are the connections used with port 8884:

```
const options = {
    port: 8884,                 // Port for encrypted connection
    host: 'test.mosquitto.org', // Mosquitto broker URL
    clientId: '1234',           // Unique client ID
    protocol: 'mqtts',          // Encrypted connection protocol
    ca: [ca],                   // Root CA cert. for encryption
    cert: [cert],               // Client certificate
    key: [key],                 // Client key
};
```

The MQTT broker `test.mosquitto.org` provided by Mosquitto can be used solely for testing purposes and to understand the behavior of this protocol.

4.3.3 Mosquitto Broker with Docker

To create a production infrastructure, the Mosquitto broker can be installed via Docker on a server [21]. The figure 4.10 represents the complete infrastructure using open-source components. Two containers were created in a Docker environment: a container for the Mosquitto MQTT Broker and a container for Redis. The two Node.js scripts connect to the containers through different ports.

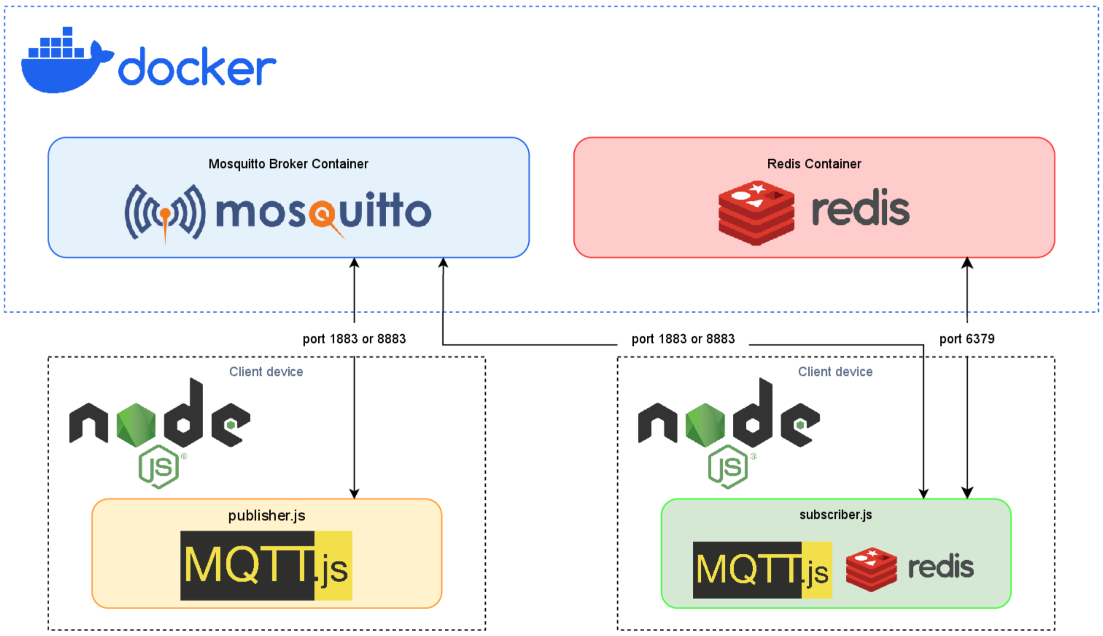


Figure 4.10: Mosquitto Broker running with Docker container

To execute the container, a Mosquitto configuration file named `mosquitto.conf` needs to be created, wherein the access ports to the server and the authentication type are defined [26] [37]. Subsequently, to create and execute the container, a `docker-compose.yml` file is required. This file defines various parameters to start the container, including port mapping to make the service accessible outside Docker [22]. To set the different authentication methods, both the container and

broker configuration files will need to be modified. The `docker-compose.yml` file also contains the information to start a redis container, in which messages received from the subscriber will be saved.
Using the following command, the container can be created and executed:

```
sudo docker-compose -p mqtt5 up -d
```

As tested with the test broker, the first authentication method is via username and password, setting port 1883. Credential management in this case occurs through the creation of a `pwfile` to save usernames and passwords. For this purpose, in the Mosquitto configuration (see appendix 1.6), the path to the password file is added using the parameter `password_file`:

```
password_file /mosquitto/config/pwfile
```

The `pwfile` can be modified by accessing the container when it is running and using the `mosquitto_passwd` command to create a user (e.g., `demo`) and prompt for the password (e.g., `demo`) [36]:

```
mosquitto_passwd -c /mosquitto/config/pwfile demo
```

The port 1883 is also specified both in the configuration and in the `docker-compose.yml` file.
It is necessary to restart the container to apply the changes made. Using the same script to access the test broker, it's possible to modify the authentication options for both the publisher and the subscriber to access the running broker via Docker.
In this case, the container is executed with Docker Desktop using a Windows environment, and the address to connect to the service is `localhost`:

```
const options = {
    port: 1883,
    host: 'localhost',
    protocol: 'mqtt',
    username: 'demo',       // MQTT broker username
    password: 'demo',       // MQTT broker password
}
```

The second authentication method is through the use of certificates to identify the client, thereby creating a TLS connection [16]. Port 8883 is used in this case. Unlike the certificates used with the test server, in this case, they need to be created:

1. As a first step, the local Authority certificate is created:

   ```
   openssl req -new -x509 -days 365 \
       -extensions v3_ca \
       -keyout ca.key \
       -out ca.crt
   ```

2. Next, a private key used by the MQTT server is generated:

   ```
   openssl genrsa -out server.key 2048
   ```

3. A certificate signing request (.csr) is created using the newly generated key, and then it is signed with the CA created in the first step:

   ```
   openssl req -out server.csr -key server.key -new
   ```

4. Finally, the request is signed to create the certificate used during authentication:

   ```
   openssl x509 -req -in server.csr \
       -CA ca.crt \
       -CAkey ca.key \
       -CAcreateserial \
       -out server.crt -days 180
   ```

The certificate just created is used to identify the Mosquitto server. To subsequently connect to the broker, it's necessary to create certificates for the clients by repeating the described steps, always using the same CA. Each client will require a specific certificate. All certificates are in PEM format.

In this case, the port used to connect is 8883, and it is important to specify the protocol for each port used in the Mosquitto conf file:

```
listener 8883
protocol mqtt
```

When the certificates are available, the Mosquitto config file is modified by adding the certificate paths (see appendix 1.7):

```
cafile my-ca.crt
certfile server.crt
keyfile server.key
```

The `docker-compose.yml` file (see appendix 1.8) is also modified by adding port 8883. The connection options are similar to the previous ones but with the addition of the specific client certificates:

```
const ca = await fsp.readFile('./mosquitto.org.crt');
const cert = await fsp.readFile('./mosquitto-cert/client.crt');
const key = await fsp.readFile('./mosquitto-cert/client.key');
const options = {
    port: 8883,              // Port for encrypted connection
    host: 'localhost',       // Mosquitto broker URL
    protocol: 'mqtts',       // Encrypted connection protocol
    ca: [ca],                // Root CA certificate for encryption
    cert: [cert],            // client certificate
    key: [key],              // client key
};
```

In all the previous cases, when the subscriber receives a message, it writes the message in Redis. The Redis client is initialized at the start of the script using a password for authentication. Using the "redis" npm package, it is possible to create a client that connects to the Redis container via REDIS_PASSWORD (default port 6379):

```
const clientRedis = await createClient({ password: REDIS_PASSWORD })
```

The received message is added to a queue with the same name as the topic ("wind") via the "lPush" method:

```
const redisResp = await clientRedis.lPush(topic, message);
```

By using these approaches, it is possible to implement an Edge infrastructure that communicates securely using open-source frameworks.

4.4 Improvements and Vulnerabilities

In both prototypes, X.509 certificates were used to securely authenticate various devices using the MQTT protocol, representing the currently safest method of authentication. However, this type of certificate can have weak points that may relate to certificate management:

- Failure to revoke and replace compromised or expired certificates
- Compromised Certificate Authorities (CAs) in case of which attackers can issue fraudulent certificates leading to man-in-the-middle attacks
- Insecure management of private keys, especially when they are inadequately protected

Especially for the prototype using Mosquitto, it is advisable to implement an adequate certificate revocation mechanism. In AWS, certificate management is well-managed.
For both solutions, it is necessary to devise a way to adequately protect the private keys of the certificates as well as the AWS access keys and Redis password.

Keeping the installed software on the devices up-to-date is also important to address breaches and vulnerabilities, and, in many cases, to improve software performance. For example, in the AWS prototype, Greengrass Version 2 was used, adding various features such as modular software components and continuous deployments compared to Version 1, simplifying the development and management of edge applications.

Chapter 5

Conclusions

In the continually evolving realm of edge and fog networks, security remains a paramount concern, particularly when it comes to developing a reliable authentication process that can be implemented on resource-constrained IoT devices.

Throughout this paper, two solutions for implementing authentication methods in edge and fog infrastructures have been presented. By mainly delving into the MQTT protocol with authentication methods based on X.509 certificate, an authentication system characterized by robustness and ease of implementation has been designed. It retains this quality even in the presence of resource-limited devices.

Prototypes have been built using widely accepted technologies and programming languages. The first prototype was constructed with assistance from Amazon Web Services (AWS), specifically AWS IoT Core, AWS Greengrass and DynamoDB. These services enable the intuitive creation and management of fog infrastructures while simultaneously ensuring network security. The second prototype was implemented using a fully open-source solution, operating the MQTT broker Mosquitto and Redis database within a Docker environment and simulating IoT devices using Node.js in conjunction with the MQTT.js package.

Testing and verification of the effectiveness and efficiency of these methods, as well as consideration of the vulnerabilities to be addressed, have demonstrated that these proposals can be successfully applied in real-edge systems. These solutions provide a secure and easy-to-implement infrastructure for securely sharing information among the various devices within the network.

Appendices

.1 Code sample

.1.1 com.example.clientdevices.MyHelloWorldSubscriber-1.1.0.json

```
{
    "RecipeFormatVersion": "2020-01-25",
    "ComponentName":
     ↪  "com.example.clientdevices.MyHelloWorldSubscriber",
    "ComponentVersion": "1.0.0",
    "ComponentDescription": "A component that subscribes to Hello
     ↪  World messages from client devices.",
    "ComponentPublisher": "Amazon",
    "ComponentConfiguration": {
      "DefaultConfiguration": {
        "accessControl": {
          "aws.greengrass.ipc.pubsub": {

             ↪  "com.example.clientdevices.MyHelloWorldSubscriber:pubsub:1":
             ↪  {
              "policyDescription": "Allows access to subscribe to
               ↪  all topics.",
              "operations": [
                "aws.greengrass#SubscribeToTopic"
              ],
              "resources": [
                "*"
              ]
            }
          }
        }
      }
    },
    "Manifests": [
      {
        "Platform": {
          "os": "linux"
        },
        "Lifecycle": {
          "install": "python3 -m pip install --user awsiotsdk
           ↪  boto3",
          "run": "python3 -u
           ↪  {artifacts:path}/hello_world_subscriber.py",
          "Setenv": {
            "aws_access_key_id": "***",
            "aws_secret_access_key": "***"
          }
        }
      },
      {
        "Platform": {
          "os": "windows"
        },
        "Lifecycle": {
          "install": "py -3 -m pip install --user awsiotsdk",
          "run": "py -3 -u
           ↪  {artifacts:path}/hello_world_subscriber.py"
        }
      }
    ]
  }
```

.1.2 hello_world_subscriber.py

```python
import sys
import time
import traceback
import boto3
import datetime
import os
import json

from awsiot.greengrasscoreipc.clientv2 import
↪  GreengrassCoreIPCClientV2
from datetime import datetime

CLIENT_DEVICE_HELLO_WORLD_TOPIC = 'clients/+/hello/world'
TIMEOUT = 10

dynamodb = boto3.resource('dynamodb')
client = boto3.client('dynamodb',
    aws_access_key_id=os.environ['aws_access_key_id'],
    aws_secret_access_key=os.environ['aws_secret_access_key'],
    )


def on_hello_world_message(event):
    try:
        message = str(event.binary_message.message, 'utf-8')
        time = str(datetime.now())

        # update dynamodb table
        client.put_item(TableName='hello-world-table',
            Item={
                'message': {'S':
                ↪  str(json.loads(message)['message'])},
                'timestamp': {'S': time},
                'topic': {'S': CLIENT_DEVICE_HELLO_WORLD_TOPIC},
                'sequence' : { 'S':
                 ↪  str(json.loads(message)['sequence'])}
            }
            )
        print('Received new message: %s' % message)
    except:
        traceback.print_exc()


try:
    ipc_client = GreengrassCoreIPCClientV2()

    # SubscribeToTopic returns a tuple with the response and the
     ↪  operation.
    _, operation = ipc_client.subscribe_to_topic(
        topic=CLIENT_DEVICE_HELLO_WORLD_TOPIC,
         ↪  on_stream_event=on_hello_world_message)
    print('Successfully subscribed to topic: %s' %
          CLIENT_DEVICE_HELLO_WORLD_TOPIC)

    # Keep the main thread alive, or the process will exit.
    try:
        while True:
            time.sleep(10)
    except InterruptedError:
        print('Subscribe interrupted.')

    operation.close()
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

.1.3 basic_discovery.py

```python
# Copyright Amazon.com, Inc. or its affiliates. All Rights
↪ Reserved.
# SPDX-License-Identifier: Apache-2.0.

import time
import json
from awscrt import io, http
from awscrt.mqtt import QoS
from awsiot.greengrass_discovery import DiscoveryClient
from awsiot import mqtt_connection_builder

from utils.command_line_utils import CommandLineUtils

allowed_actions = ['both', 'publish', 'subscribe']

# cmdData is the arguments/input from the command line placed
↪ into a single struct for
# use in this sample. This handles all of the command line
↪ parsing, validating, etc.
# See the Utils/CommandLineUtils for more information.
cmdData = CommandLineUtils.parse_sample_input_basic_discovery()

tls_options = io.TlsContextOptions
    .create_client_with_mtls_from_path(
        cmdData.input_cert,
        cmdData.input_key
    )

if (cmdData.input_ca is not None):
    tls_options.override_default_trust_store_from_path(None,
    ↪ cmdData.input_ca)
tls_context = io.ClientTlsContext(tls_options)

socket_options = io.SocketOptions()

```

```python
proxy_options = None
if cmdData.input_proxy_host is not None and cmdData.input_proxy_port != 0:
    proxy_options = http.HttpProxyOptions(cmdData.input_proxy_host, cmdData.input_proxy_port)

print('Performing greengrass discovery...')
discovery_client = DiscoveryClient(
    io.ClientBootstrap.get_or_create_static_default(),
    socket_options,
    tls_context,
    cmdData.input_signing_region, None, proxy_options)
resp_future = discovery_client.discover(cmdData.input_thing_name)
discover_response = resp_future.result()

if (cmdData.input_is_ci):
    print("Received a greengrass discovery result! Not showing result in CI for possible data sensitivity.")
else:
    print(discover_response)

if (cmdData.input_print_discovery_resp_only):
    exit(0)


def on_connection_interupted(connection, error, **kwargs):
    print('connection interrupted with error {}'.format(error))


def on_connection_resumed(connection, return_code, session_present, **kwargs):
    print('connection resumed with return code {}, session present {}'.format(return_code, session_present))


```

```
# Try IoT endpoints until we find one that works
def try_iot_endpoints():
    for gg_group in discover_response.gg_groups:
        for gg_core in gg_group.cores:
            for connectivity_info in gg_core.connectivity:
                try:
                    print(
                        f"Trying core {gg_core.thing_arn} at host
                        ↪  {connectivity_info.host_address} port
                        ↪  {connectivity_info.port}")
                    mqtt_connection = mqtt_connection_builder
                        .mtls_from_path(
                        endpoint=connectivity_info.host_address,
                        port=connectivity_info.port,
                        cert_filepath=cmdData.input_cert,
                        pri_key_filepath=cmdData.input_key,
                        ca_bytes=
                            gg_group.certificate_authorities[0]
                            .encode('utf-8'),
                        on_connection_interrupted=
                            on_connection_interupted,
                        on_connection_resumed=
                            on_connection_resumed,
                        client_id=cmdData.input_thing_name,
                        clean_session=False,
                        keep_alive_secs=30)

                    connect_future = mqtt_connection.connect()
                    connect_future.result()
                    print('Connected!')
                    return mqtt_connection

                except Exception as e:
                    print('Connection failed with exception
                    ↪  {}'.format(e))
                    continue
```

```python

    exit('All connection attempts failed')


mqtt_connection = try_iot_endpoints()

if cmdData.input_mode == 'both' or cmdData.input_mode == 'subscribe':
    def on_publish(topic, payload, dup, qos, retain, **kwargs):
        print('Publish received on topic {}'.format(topic))
        print(payload)
    subscribe_future, _ = mqtt_connection.subscribe(cmdData.input_topic, QoS.AT_MOST_ONCE, on_publish)
    subscribe_result = subscribe_future.result()

loop_count = 0
while loop_count < cmdData.input_max_pub_ops:
    if cmdData.input_mode == 'both' or cmdData.input_mode == 'publish':
        message = {}
        message['message'] = cmdData.input_message
        message['sequence'] = loop_count
        messageJson = json.dumps(message)
        pub_future, _ = mqtt_connection.publish(cmdData.input_topic, messageJson, QoS.AT_MOST_ONCE)
        pub_future.result()
        print('Published topic {}: {}\n'.format(cmdData.input_topic, messageJson))

        loop_count += 1
    time.sleep(1)
```

.1.4 mqtt-wind-publisher.js

```
/**
 * The mqtt-wind-publisher script connects to the mqtt broker and
 ↪ publishes messages to the "wind" topic.
 *
 * Usage: Run the script within a Node.js environment passing the
 ↪ connection option for the MQTT broker as an argument:
 * - option_1 --> Mosquitto test broker using username and
 ↪ password
 * - option_2 --> Mosquitto test broker using X.509 certificates
 * - option_3 --> Mosquitto broker running on localhost (Docker
 ↪ container) using username and password
 * - option_4 --> Mosquitto broker running on localhost (Docker
 ↪ container) using X.509 certificates
*/

const mqtt = require('async-mqtt');
const fsp = require('fs').promises;

(async () => {
    try {
        // sample wind parameter
        const sampleData = [
            {
              ID_stazione: '012345',
              stazione: 'Location_1',
              data: '2023-12-03 12:00:00',
              valore: '32 Gradi, 3.5 m/s, 4.1 m/s'
            },
        ]

        // read connection option argument
        const inputOption = process.argv[2];

        // define connection option variable
        let options = undefined;

        // read certificates
        const caTest = await fsp.readFile('./mosquitto.org.crt');
        const ca = await fsp.readFile('./mosquitto-cert/ca.crt');
        const cert = await
         ↪  fsp.readFile('./mosquitto-cert/client.crt');
        const key = await
         ↪  fsp.readFile('./mosquitto-cert/client.key');

        // ------ test.mosquitto.org ------
        // Define the connection options
        // 8885 : MQTT, encrypted, authenticated
        if (inputOption === 'option_1') {
            options = {
                port: 8885, // Port for encrypted connection
                host: 'test.mosquitto.org', // Mosquitto broker
                 ↪  URL
                clientId: '1234', // Unique client ID
                username: 'rw', // MQTT broker username
                password: 'readwrite', // MQTT broker password
                protocol: 'mqtts', // Encrypted connection
                 ↪  protocol
                ca: [caTest], // Root CA certificate for
                 ↪  encryption
            };
        }

        // 8884 : MQTT, encrypted, client certificate required
        if (inputOption === 'option_2') {
            options = {
                port: 8884, // Port for encrypted connection
                host: 'test.mosquitto.org', // Mosquitto broker
                 ↪  URL
                clientId: '1234', // Unique client ID
                protocol: 'mqtts', // Encrypted connection
                 ↪  protocol
                ca: [caTest], // Root CA certificate for
                 ↪  encryption
                cert: [cert], // client certificate
                key: [key], // client key
            };
        }

        // Create test client
        // const client = await
         ↪  mqtt.connectAsync('mqtt://test.mosquitto.org');

        // ------ Container Mosquitto ------
        // 1883: username demo, password demo
        if (inputOption === 'option_3') {
            options = {
              port: 1883,
              host: 'localhost',
              protocol: 'mqtt',
              username: 'demo', // MQTT broker username
              password: 'demo', // MQTT broker password
            }
        }

        // 8883: certificates
        if (inputOption === 'option_4') {
            options = {
              port: 8883, // Port for encrypted connection
              host: 'localhost', // Mosquitto broker URL
              protocol: 'mqtts', // Encrypted connection protocol
              ca: [ca], // Root CA certificate for encryption
              cert: [cert], // client certificate
              key: [key], // client key
              keepalive: 1,
              connectTimeout: 3 * 1000,
            };
        }

        if(options){
            // Create the MQTT client
            const client = await mqtt.connectAsync(options);

            // 1. publish the message
            for (let i = 0; i < sampleData.length; i++) {
              const obj = sampleData[i];
              await client.publish('wind', JSON.stringify(obj), {
               ↪  retain: true });
            }

            // disconnect from the broker
            client.on('disconnect', (message) => {
              console.log('disconnect', message);
            })

            // close the connection
            client.on('close', () => {
              console.log('done');
              return;
            })
        } else {
            console.log('Invalid connection option entered');
            return;
        }
    } catch (e) {
        console.log(e);
        process.exit(1);
    }
})();
```

.1.5 mqtt-wind-subscriber.js

```
/**
 * The mqtt-wind-subscriber script connects to the mqtt broker and receives messages published in the "wind" topic.
 * The received messages are then saved in the Redis database.
 *
 * Usage: Run the script within a Node.js environment passing the connection option for the MQTT broker as an argument:
 * - option_1 --> Mosquitto test broker using username and password
 * - option_2 --> Mosquitto test broker using X.509 certificates
 * - option_3 --> Mosquitto broker running on localhost (Docker container) using username and password
 * - option_4 --> Mosquitto broker running on localhost (Docker container) using X.509 certificates
*/

require('dotenv').config();
// retrieve environment variables
const REDIS_PASSWORD = process.env.REDIS_PASSWORD;

const mqtt = require('async-mqtt');
const fsp = require('fs').promises;

// init Redis client
const { createClient } = require('redis');

(async () => {
    try {
        // read connection option argument
        const inputOption = process.argv[2];

        // define connection option variable
        let options = undefined;

        // read certificates
        const caTest = await fsp.readFile('./mosquitto.org.crt');
        const ca = await fsp.readFile('./mosquitto-cert/ca.crt');
        const cert = await fsp.readFile('./mosquitto-cert/client.crt');
        const key = await fsp.readFile('./mosquitto-cert/client.key');

        // connection redis in localhost:6379
        const clientRedis = await createClient({ password: REDIS_PASSWORD })
          .on('error', err => console.log('Redis Client Error', err))
          .connect();

        // ------ test.mosquitto.org ------
        // Define the connection options
        // 8885 : MQTT, encrypted, authenticated
        if (inputOption === 'option_1') {
            options = {
                port: 8885, // Port for encrypted connection
                host: 'test.mosquitto.org', // Mosquitto broker URL
                clientId: '1234', // Unique client ID
                username: 'rw', // MQTT broker username
                password: 'readwrite', // MQTT broker password
                protocol: 'mqtts', // Encrypted connection protocol
                ca: [caTest], // Root CA certificate for encryption
            };
        }

        // 8884 : MQTT, encrypted, client certificate required
        if (inputOption === 'option_2') {
            options = {
                port: 8884, // Port for encrypted connection
                host: 'test.mosquitto.org', // Mosquitto broker URL
                clientId: '1234', // Unique client ID
                protocol: 'mqtts', // Encrypted connection protocol
                ca: [caTest], // Root CA certificate for encryption
                cert: [cert], // client certificate
                key: [key], // client key
            };
        }

        // Create test client
        // const client = await mqtt.connectAsync('mqtt://test.mosquitto.org');

        // ------ Container Mosquitto ------
        // 1883: username demo, password demo
        if (inputOption === 'option_3') {
            options = {
              port: 1883,
              host: 'localhost',
              protocol: 'mqtt',
              username: 'demo', // MQTT broker username
              password: 'demo', // MQTT broker password
            }
        }

        // 8883: certificates
        if (inputOption === 'option_4') {
            const options = {
              port: 8883, // Port for encrypted connection
              host: 'localhost', // Mosquitto broker URL
              protocol: 'mqtts', // Encrypted connection protocol
              ca: [ca], // Root CA certificate for encryption
              cert: [cert], // client certificate
```

```
                key: [key], // client key
                keepalive: 1,
                connectTimeout: 3 * 1000,
            };
        }

        if(options){
            // Create the MQTT client
            const client = await mqtt.connectAsync(options);

            // 1. subscribe to a topic
            await client.subscribe('wind');

            // 3. read message
            client.on('message', async (topic, message) => {
              console.log('topic "' + topic + '": ' +
              ↪  message.toString());

              // write message in Redis
              const redisResp = await clientRedis.lPush(topic,
              ↪  message);
              console.log("add key to redis", redisResp);
              // get current element in key topic
              const currentMessages = await
              ↪  clientRedis.lRange(topic, 0, -1);
              console.log("current elements: ", currentMessages);
            });

            // disconnect from the broker
            client.on('disconnect', async (message) => {
              console.log('disconnect', message);

              // disconnect redis
              await clientRedis.disconnect();
            })

            // close the connection
            client.on('close', () => {
              console.log('done');
              return;
            })
        } else {
            console.log('Invalid connection option entered');

            // disconnect redis
            await clientRedis.disconnect();

            return;
        }
    } catch (e) {
        console.log(e);
        process.exit();
    }
})();
```

.1.6 mosquitto.conf with pwfile

```
listener 1883
protocol mqtt

allow_anonymous false
persistence true
password_file /mosquitto/config/pwfile
persistence_file mosquitto.db
persistence_location /mosquitto/data/
```

.1.7 mosquitto.conf with certificates

```
listener 8883
protocol mqtt

cafile /mosquitto/config/ca.crt
certfile /mosquitto/config/server.crt
keyfile /mosquitto/config/server.key

require_certificate true
use_identity_as_username true
use_subject_as_username true
persistence true
allow_anonymous false

persistence_file mosquitto.db
persistence_location /mosquitto/data/
```

.1.8 docker-compose.yml

```yaml
version: "3.7"
services:
  # mqtt5 eclipse-mosquitto
  mqtt5:
    image: eclipse-mosquitto
    container_name: mosquitto
    ports:
      # - "1883:1883" #default mqtt port for username-password
      - "8883:8883" #default mqtt port for certificates
      - "9001:9001" #default mqtt port for websockets
    volumes:
      - ./config:/mosquitto/config:rw
      - ./data:/mosquitto/data:rw
      - ./log:/mosquitto/log:rw
  # redis
  redis:
    image: redis
    container_name: redis
    ports:
      - "6379:6379"
    volumes:
      - ./redis/redis.conf:/usr/local/etc/redis/redis.conf
    command: ["redis-server", "/usr/local/etc/redis/redis.conf"]
# volumes for mapping data,config and log
volumes:
  config:
  data:
  log:
networks:
  default:
    name: mqtt5-network
```

Bibliography

[1] *A 2022 Guide to IoT Protocols and Standards.* URL: https://www.particle.io/iot-guides-and-resources/iot-protocols-and-standards/.

[2] AWS Amazon. *Authentication.* URL: https://docs.aws.amazon.com/iot/latest/developerguide/authentication.html.

[3] AWS Amazon. *Boto3 1.28.9 Documentation - Credentials.* URL: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/credentials.html.

[4] AWS Amazon. *Create a virtual device with Amazon EC2.* URL: https://docs.aws.amazon.com/iot/latest/developerguide/creating-a-virtual-thing.html.

[5] AWS Amazon. *Device authentication and authorization for AWS IoT Greengrass.* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/device-auth.html.

[6] AWS Amazon. *Device authentication and authorization for AWS IoT Greengrass.* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/device-auth.html.

[7] AWS Amazon. *Getting started - Step 2: Set up your environment.* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/getting-started-set-up-environment.html.

[8] AWS Amazon. *Getting started - Step 4: Develop and test a component on your device.* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/create-first-component.html.

[9] AWS Amazon. *Install the AWS IoT Greengrass Core software (console).* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/install-greengrass-v2-console.html.

[10] AWS Amazon. *Interact with local IoT devices.* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/interact-with-local-iot-devices.html.

[11] AWS Amazon. *Tutorial: Interact with local IoT devices over MQTT.* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/client-devices-tutorial.html.

[12] AWS Amazon. *What is Amazon DynamoDB?* URL: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html.

[13] AWS Amazon. *What is AWS IoT Greengrass?* URL: https://docs.aws.amazon.com/greengrass/v2/developerguide/what-is-iot-greengrass.html.

[14] AWS Amazon. *What is AWS IoT?* URL: https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html.

[15] AWS Amazon. *X.509 client certificates.* URL: https://docs.aws.amazon.com/iot/latest/developerguide/x509-client-certs.html.

[16] Ralight on asciinema. *Generating a TLS certificate for mosquitto.* URL: https://asciinema.org/a/201826.

[17] *async-mqtt.* URL: https://www.npmjs.com/package/async-mqtt.

[18] *Authentication in Microservices: Approaches and Techniques.* 2022. URL: https://frontegg.com/blog/authentication-in-microservices.

[19] John M. Acken; Naresh K. Sehgal; Divya Bansal; Robert B. Bass. *Security and Trust Metrics for Edge Computing.* 2023. URL: https://ieeexplore.ieee.org/abstract/document/10102745.

[20] Ellen Boehm. *The Top IoT Authentication Methods and Options.* 2020. URL: https://www.keyfactor.com/blog/the-top-iot-authentication-methods-and-options/.

[21] Docker. *eclipse-mosquitto.* URL: https://hub.docker.com/_/eclipse-mosquitto.

[22] Docker. *Networking overview.* URL: https://docs.docker.com/network/.

[23] *Edge computing.* URL: https://en.wikipedia.org/wiki/Edge_computing.

[24] *Fog computing.* URL: https://en.wikipedia.org/wiki/Fog_computing.

[25] FromWikipediathefreeencyclopedia. *MQTT*. URL: https://en.wikipedia.org/wiki/MQTT.

[26] *How to setup Mosquitto MQTT Broker using docker*. URL: https://github.com/sukesh-ak/setup-mosquitto-with-docker.

[27] *HTTP authentication*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication.

[28] *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. URL: https://datatracker.ietf.org/doc/html/rfc5280.

[29] *Introduction to MQTT Security Mechanisms*. URL: http://www.steves-internet-guide.com/mqtt-security-mechanisms/.

[30] Kashif Naseer Qureshi; Gwanggil Jeon; Mohammad Mehedi Hassan; Md. Rafiul Hassan; Kuljeet Kaur. *Blockchain-Based Privacy-Preserving Authentication Model Intelligent Transportation Systems*. 2023. URL: https://ieeexplore.ieee.org/document/9745465.

[31] Esther Villar-Rodriguez; María Arostegi Pérez; Ana I. Torre-Bastida; Cristina Regueiro Senderos; Juan López-de-Armentias. *Edge intelligence secure frameworks: Current state and future challenges*. 2023. URL: https://www.sciencedirect.com/science/article/pii/S0167404823001888.

[32] Milan Milenkovic. *Internet of things: concepts and system design*. 2020.

[33] *Monolithic application*. URL: https://en.wikipedia.org/wiki/Monolithic_application.

[34] Eclipse Mosquitto™. *Eclipse Mosquitto*. URL: https://mosquitto.org/.

[35] Eclipse Mosquitto™. *Generate a TLS client certificate for test.mosquitto.org*. URL: https://test.mosquitto.org/ssl/.

[36] Eclipse Mosquitto™. *$mosquitto_passwdmanpage$*. URL: https://mosquitto.org/man/mosquitto_passwd-1.html.

[37] Eclipse Mosquitto™. *mosquitto.conf man page*. URL: https://mosquitto.org/man/mosquitto-conf-5.html.

[38] Eclipse Mosquitto™. *test.mosquitto.org*. URL: https://test.mosquitto.org/.

[39] MQTT. *MQTT: The Standard for IoT Messaging*. URL: https://mqtt.org/.

[40] Anuj Pundalik. *From legacy systems to microservices: Transforming auth architecture*. 2023. URL: https://www.contentstack.com/blog/tech-talk/from-legacy-systems-to-microservices-transforming-auth-architecture.

[41] *Redis*. URL: https://redis.io/.

[42] *redis - npm*. URL: https://www.npmjs.com/package/redis.

[43] Talal Halabi Saud Al Harbi and Martine Bellaiche. *Fog Computing Security Assessment for Device Authentication in the Internet of Things*. 2020. URL: https://www.researchgate.net/publication/346884900_Fog_Computing_Security_Assessment_for_Device_Authentication_in_the_Internet_of_Things.

[44] Tzachi Strugo. *Authentication & Authorization in Microservices Architecture - Part I*. 2021. URL: https://dev.to/behalf/authentication-authorization-in-microservices-architecture-part-i-2cn0.

[45] *The Future of Autonomous Cars with Edge Computing*. URL: https://www.analyticssteps.com/blogs/future-autonomous-cars-edge-computing.

[46] *Top 10 Cloud Service Providers Globally in 2023*. 2023. URL: https://dgtlinfra.com/top-cloud-service-providers.

[47] *Understanding IoT device authentication*. 2021. URL: https://mateenfaisal.medium.com/understanding-iot-authentication-bbad5fe83271.

[48] *X.509*. URL: https://en.wikipedia.org/wiki/X.509.

[49] Zexuan Luo; Shirui Zhao. *Deep Dive into Authentication in Microservices*. 2022. URL: https://api7.ai/blog/understanding-microservices-authentication-services.