DECENTRALIZED IDENTIFIERS FOR THE INTERNET OF THINGS

Konstantinos Betchavas

Supervisors: Prof. George Xylomenos, Dr. Nikos Fotiou

Master's Thesis

Athens University of Economics and Business

2022

Περίληψη

Αυτή η εργασία εξετάζει τη χρήση Αποκεντρωμένων Αναγνωριστικών (DIDs) στο Διαδίκτυο των Πραγμάτων. Συγκεκριμένα εξετάζουμε την χρήση τους για αυθεντικοποίηση δεδομένων που λαμβάνουμε από συσκευές. Στην καθημερινή μας ζωή βασιζόμαστε σε αναγνωριστικά, όπως διευθύνσεις email και ονόματα χρήστη, τα οποία εκδίδονται και ελέγχονται από εξωτερικές αρχές και δεν είναι υπό τον έλεγχό μας. Το Αποκεντρωμένο Αναγνωριστικό (DID) είναι ένας νέος τύπος αναγνωριστικού που επιτρέπει επαληθεύσιμες, αποκεντρωμένες ψηφιακές ταυτότητες, που μας δίνει πίσω τον έλεγχο στην ταυτότητά μας, καταργώντας την ανάγκη για μια συγκεντρωτική εξωτερική αρχή. Τα DIDs μπορούν επίσης να χρησιμοποιηθούν από συσκευές που αποτελούν μέρος του Διαδίκτυου των Πραγμάτων. Τα DIDs χρησιμοποιούνται σε διάφορες εφαρμογές ασφαλείας, με πιο σημαντική την αυθεντικοποίηση.

Στην εργασία μας χρησιμοποιήσαμε μία μέθοδο δημιουργίας DIDs, τη μέθοδο did:self, που δημιουργήθηκε στο Οικονομικό Πανεπιστήμιο Αθηνών, ειδικά για το Διαδίκτυο των Πραγμάτων. Οι ιδιοκτήτες των did:self DIDs είναι υπεύθυνοι για τη δημιουργία και τη διάδοση των εγγράφων DID από μόνοι τους.

Υπάρχουν πολλά εργαλεία που μπορούν να χρησιμοποιηθούν σε περιβάλλον IoT. Το πρωτόκολλο CoAP είναι σημαντικό γιατί είναι ιδανικό για χρήση σε περιβάλλοντα με περιορισμένους πόρους. Υπάρχουν πολλοί τρόποι για να τοποθετήσετε, να ρυθμίσετε τις συσκευές σας και να ενεργοποιήσετε την επικοινωνία μαζί τους. Τα περισσότερα από αυτά περιλαμβάνουν μια πύλη IoT, η οποία βρίσκεται ανάμεσα στις συσκευές και το Διαδίκτυο. Το RIOT είναι ένα λειτουργικό σύστημα, που δημιουργήθηκε ειδικά για την υποστήριξη του Διαδικτύου των Πραγμάτων. Προσφέρει πολλά πλεονεκτήματα στους προγραμματιστές και είναι αρκετά ελαφρύ ώστε να μπορεί να εκτελεστεί σε πολύ απλές συσκευές. Το χρησιμοποιούμε για την υλοποίηση DIDs με τη μέθοδο did:self.

Η εργασία μας δείχνει ότι μπορούμε να χρησιμοποιήσουμε Αποκεντρωμένα Αναγνωριστικά, τα οποία δημιουργούνται και κοινοποιούνται από τις συσκευές στο IoT. Από το λειτουργικό σύστημα RIOT χρησιμοποιούνται διάφορες βιβλιοθήκες, που μας βοηθούν να εφαρμόσουμε τη μέθοδο did:self, γραμμένη σε γλώσσα προγραμματισμού C,

στις συσκευές. Στα πλαίσια της εργασίας μας μεταφέρθηκαν πρόσθετες βιβλιοθήκες και κώδικας στο RIOT ώστε να μπορούμε να δημιουργούμε και να επαληθεύουμε αποκεντρωμένα αναγνωριστικά με βάση τη μέθοδο did:self.

Προστέθηκε κώδικας που χρησιμοποιεί τις βιβλιοθήκες αυτές για να δημιουργήσουμε κλειδιά EdDSA, να υπογράψουμε δεδομένα με αυτά και να κωδικοποιήσουμε δεδομένα ως base64url. Επιπλέον, δημιουργήθηκαν οι κατάλληλες συναρτήσεις για την δημιουργία και αποθήκευση των DID Document και DID Proof στις συσκευές. Αυτά απαιτούνται για να αυθεντικοποιήσουμε πρώτα το DID Document και αργότερα με την χρήση αυτού, οποιαδήποτε δεδομένα λαμβάνουμε από τις συσκευές. Στον κώδικα μας επιβεβαιώνουμε την αυθεντικοποίηση των πόρων που λαμβάνουμε από τις συσκευές στο Gateway που δημιουργήσαμε.

Επομένως, τα Αποκεντρωμένα Αναγνωριστικά (DIDs) βοηθούν με επιτυχία στην διασφάλιση της αυθεντικότητας των δεδομένων που λαμβάνουμε από τις συσκευές. Στο 6^{o} κεφάλαιο δείχνουμε παραδείγματα της χρήσης τους στο RIOT, αναφέροντας τα αποτελέσματα που λαμβάνουμε.

Abstract

This thesis examines the use of Decentralized Identifiers (DIDs) in the Internet of Things. In particular we are examining their use for authenticating the data that we receive from devices. In our daily lives we rely on identifiers, such as email addresses and usernames, which are issued and controlled by external authorities and are not under our control. A Decentralized Identifier (DID) is a new type of identifier that enables verifiable, decentralized digital identities, giving us back control over our identity by removing the need for a centralized external authority. DIDs can also be used by devices that are part of the Internet of Things. DIDs are used in various security applications, the most important being authentication.

In our work we used a method of generating DIDs, the did:self method, created at the Athens University of Economics and Business, specifically for the Internet of Things. Owners of did:self DIDs are responsible for creating and propagating DID documents themselves.

There are many tools that can be used in an IoT environment. The CoAP protocol is important, because it is ideal for resource-constrained environments. There are many ways to setup and configure your IoT devices and enable communication with them. Most of them include an IoT gateway, placed between the devices and the Internet. RIOT is an operating system, built specifically to support the Internet of Things. It offers many advantages to developers and is lightweight enough to run on very simple devices. We use it to implement DIDs with the did:self method.

Our work shows that we can use Decentralized Identifiers, which are generated and shared by the devices in the IoT. Various libraries are used by the RIOT OS, which help us implement the did:self method, written in C programming language, on the devices. As part of our work we ported additional libraries and code to RIOT so that we can create and verify decentralized IDs based on the did:self method.

Added code that uses these libraries to generate EdDSA keys, sign data with them and to base64url encode data. In addition, the appropriate functions were made to create and store the DID Document and DID Proof on the devices. These are required to first authenticate the DID Document, and later using it, any data we receive from the devices. In our code we authenticate the resources we receive from the devices in the Gateway we created.

Therefore, Decentralized Identifiers (DIDs) successfully help ensure the authenticity of the data we receive from the devices. In chapter 6 we show examples of their use in RIOT, reporting the results we obtain.

Keywords: Decentralized Identifier (DID), did:self Method, Internet of Things (IoT), Constrained Application Protocol (CoAP), RIOT OS, C Programming Language.

Table of Contents

Table of Figures

1. Decentralized Identifiers (DIDs)

Individuals and organizations rely on identifiers such as email addresses, usernames and phone numbers on social media [1]. However, most identifiers are issued and controlled by external authorities, which ensure that they are globally unique. These authorities decide who or what these identifiers refer to and when they can be revoked. These identifiers are, therefore, not under our control.

DIDs are a new type of identifier that enables verifiable, decentralized digital identities. DIDs can refer to persons, data models, organizations, things, or abstract entities, as determined by a DID 'controller'. In contrast to typical identifiers, DIDs are designed so that they may be decoupled from centralized registries and identity providers. Specifically, the design allows the controller of a DID to prove control over it without requiring permission from any third party. Of course, third parties might be used to help enable the discovery of information related to a DID.

The DID specification does not enforce any technology or cryptographic method for the generation, resolution, persistence, or interpretation of DIDs. Indeed, different DID instantiations, called *methods*, can be defined, specifying how the identifiers are generated and resolved. For example, to enable interworking, implementers can create DIDs based on identifiers registered in federated or centralized identity management systems. Indeed, the majority of identifier systems can add support for DIDs. This creates an interoperability bridge between centralized, federated, and decentralized identifiers.

1.1. A Simple Example

A Decentralized Identifier is a plain text string consisting of three parts [1]:

1) the did URI scheme identifier,
2) the identifier of the DID method, and
3) the method-specific identifier of the DID.

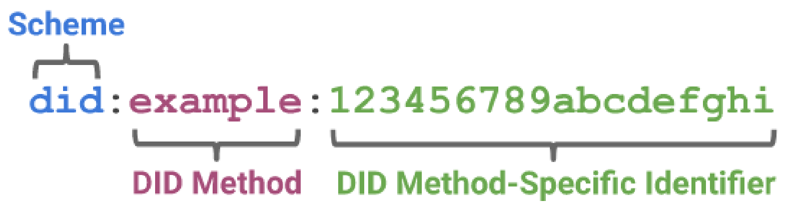

Scheme
did:example:123456789abcdefghi
DID Method
DID Method-Specific Identifier

Figure 1: An example of a DID [1]

The figure above shows an example of a DID, with its three components. DID URLs refer to a DID subject and resolve to DID Documents, which contain information associated with the DID, such as cryptographic public keys, services, and interactions.

1.2. Design Goals

The most important design goals for Decentralized Identifiers are [1]:

- **Decentralization**: Abolish the requirement for centralized authorities and avoid single points of failure in identifier management, including the registration of globally unique identifiers, public verification keys, services, and other information.
- **Control**: Give entities, both human and non-human, the power to directly control their digital identifiers without the need to rely on external authorities.
- **Privacy**: Enable entities to control the privacy of their information, including minimal, selective, and progressive disclosure of attributes or other data.
- **Security**: For their required level of assurance, enable requesting parties to depend on DID documents for sufficient security.
- **Proof-based**: Allow DID controllers to provide cryptographic proof when interacting with other entities.

1.3. Architecture Overview

We provide below a basic overview of the major components of the Decentralized Identifier architecture [1], which are schematically depicted in the following figure.

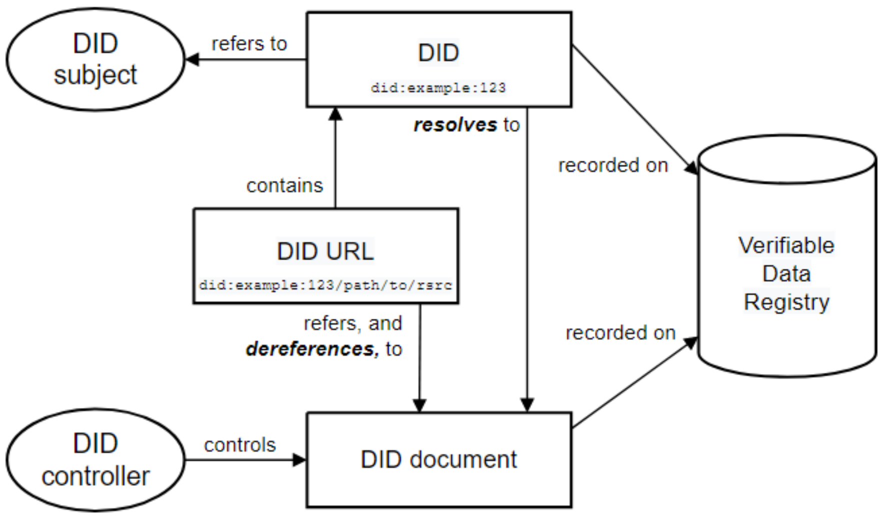


Figure 2: Overview of DID architecture and relationships of its basic components [1]

- ❖ **DIDs and DID URLs:** A Decentralized Identifier is a URI composed of three parts: the "did" scheme, a method identifier, and a unique, method-specific identifier in the format specified by the DID method. A DID URL extends the syntax of a basic DID to include other standard URI components such as path, query, and fragment, in order to locate a particular resource.
- ❖ **DID subjects:** The subject of a DID is, by definition, the entity identified by the DID. The DID subject might also be the DID controller, but this is not necessary. Anything can be the subject of a DID: a person, group, organization, thing, or concept.
- ❖ **DID controllers:** The controller of a DID is the entity (person, organization, or autonomous software) that has the ability to make changes to a DID document, as defined by a DID method. The subject of the DID may delegate control of the DID to another entity, hence the differentiation between subject and controller.
- ❖ **Verifiable data registries:** In order to be resolvable to DID documents, DIDs are typically recorded on an underlying system or network of some kind. Regardless,

of the specific technology used, any system that supports recording DIDs and returning data necessary to produce DID documents is called a verifiable data registry. Examples include distributed ledgers, decentralized file systems, and databases of any kind.

- ❖ **DID documents:** DID documents contain information associated with a DID. They typically express verification methods, such as cryptographic public keys, and services relevant for interactions with the DID subject.
- ❖ **DID methods:** DID methods are the mechanism by which a particular type of DID and its associated DID document are created, resolved, updated, and deactivated. The method specifies how exactly the identifier of a DID is formed.
- ❖ **DID resolvers and DID resolution:** A DID resolver is a system component that takes a DID as input and produces a conforming DID document as output. This process is called DID resolution.
- ❖ **DID URL dereferencers and DID URL dereferencing:** A DID URL dereferencer is a system component that takes a DID URL as input and produces a resource as output. This process is called DID URL dereferencing.

1.4. Objections

DIDs are still in the ‘proposed recommendation’ status. There are many objections to releasing the standard, as stated to W3C [2]. The main objections are:

1) Most DIDs cannot be easily memorized by humans; **they are internet friendly but not human friendly**. Many people memorize their social security number. There is no standardized way to turn such ‘human-manageable’ IDs to DIDs.
2) The documented design goal for decentralization is to ‘**eliminate the requirement for centralized authorities or single points of failure in identifier management**’. Distributed ledger technology, like a blockchain, is the best technology for this purpose, providing superior levels of distribution, programmability, and resilience. The standard itself **does not enforce decentralization on the verifiable data registry**. The objection is that this violates the stated design goals.

3) *DID methods* can be specified differently and implemented on various systems, so they are not interoperable. **Standardization of the methods was taken out of scope for version 1.0 of the recommendation.** The objection is that the proliferation of non-interoperable method specifications could drastically limit the practical use and adoption of DIDs.
4) Blockchains, if used, **can** be environmentally harmful (not all are), which is against stated W3C principles. The objection is that the standard is not actively dealing with this issue.

2. The did:self method

In this section we describe the DID method specified in “IoT Group Membership Management Using Decentralized Identifiers and Verifiable Credentials” [3], called did:self. The did:self method was created specifically for the Internet of Things and is the method we employ in this thesis.

2.1. Key Properties

As already mentioned, DID specifications allow DID method implementors to decide upon the information to be included in the DID documents of their method, as well as how an appropriate registry operates. These are left to individual DID instantiations, also referred to as DID methods.

The key features of the did:self method are:

- Owners of did:self DIDs are responsible for disseminating their DID documents by themselves, e.g., by directly transmitting them to interested parties, or by storing them in publicly accessible locations, such as a Web server. The did:self method ensures that a DID document can be verified to be correct, even if retrieved over an unsecured channel.
- did:self allows multiple valid DID documents for a specific DID to co-exist. We can take advantage of this feature to allow each IoT device to be configured with a different DID document for the same did:self DID.

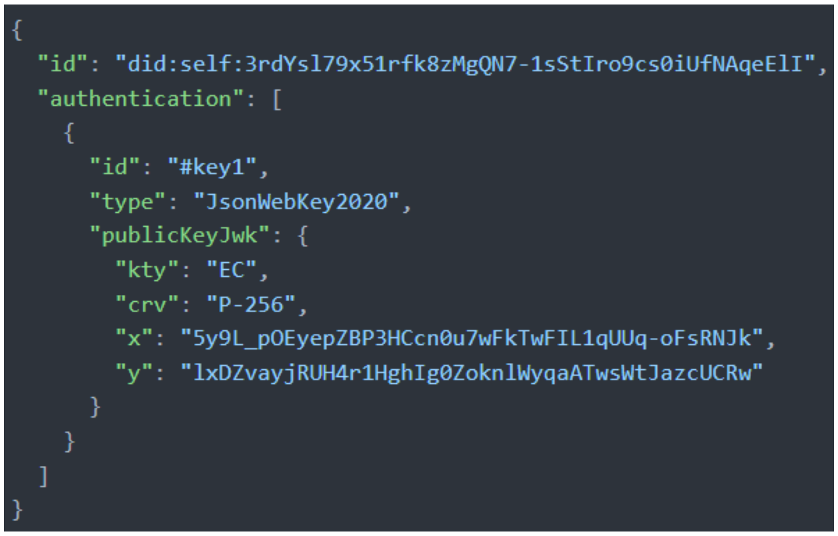
2.2. Design

A did:self based DID is a base64url encoded Ed22519 public key prefixed with the string “did:self :”. We can see an example below.

did:self:ZlFJd-4wuc7M1_6hLQRinQ7-0K5hRTr95h72ujVVzNg

A DID document in did:self may include any of the properties defined by the DID specifications, encoded using JSON. It includes the following properties, some of them optional, with the corresponding purpose:

1. **id**: The DID which the document concerns
2. **verificationMethod**: A list of public keys, with each key in the list identified by an id. There cannot be two keys with the same id for the same did:self DID, even if these keys are defined in different DID documents. We can take advantage of this property to achieve efficient authentication key rotation and detect breaches of the private key used by an authentication method. These two security properties are achieved by following a “use the most recent key” principle.
3. **authentication**: A list of public keys (or key identifiers) that can be used to authenticate the DID holder. The private key that corresponds to an authentication key is used for signing CoAP messages; hence, the corresponding public key is used for authenticating message senders.
4. **assertion**: A list of public keys (or key identifiers) that can be used to verify digital signatures of Verifiable Credentials (VCs). The private key that corresponds to an assertion key is used in our system for signing issued VCs. Hence, the corresponding public key is used for verifying these signatures. In the image below we can see the use of the P-256 Digital Signature Algorithm. For our implementation we utilize ed25519 instead.



```
{
  "id": "did:self:3rdYsl79x51rfk8zMgQN7-1sStIro9cs0iUfNAqeElI",
  "authentication": [
    {
      "id": "#key1",
      "type": "JsonWebKey2020",
      "publicKeyJwk": {
        "kty": "EC",
        "crv": "P-256",
        "x": "5y9L_pOEyepZBP3HCcn0u7wFkTwFIL1qUUq-oFsRNJk",
        "y": "lxDZvayjRUH4r1HghIg0ZoknlWyqaATwsWtJazcUCRw"
      }
    }
  ]
}
```

Figure 3: A DID document example

A proof is associated with each DID document; it is a “compact serialization” of a JSON Web Signature (JWS), used to validate the binding between a DID document and the corresponding did:self DID.

The payload of the proof includes the following claims:

1. The proof’s header includes the following claims:
    - alg: The algorithm used for generating the proof
    - jwk: The JWK that can be used for verifying the proof
2. The proof’s payload includes the following claims:
    - iat: The date and time of the proof’s generation
    - exp: An expiration time
    - s256: The base64url encoded hash of the DID document, calculated using SHA-256

The signature of the proof is generated using the private key that corresponds to the did:self DID and the Edwards-curve Digital Signature Algorithm (EdDSA).

Given a did:self DID, a DID document, and the document proof, any entity can trivially verify the binding between the DID and the document by executing the following steps:

1) Verify that the identifier is equal to the thumbprint of the jwk field of the header of proof.
2) Verify that the digest of the DID document is equal to s256 in the proof.
3) Verify that the proof has not expired.
4) Verify the proof using the jwk field of the header.

2.3. Data transfer with DIDs

DIDs can help with data authentication in many ways. In our scheme we send our data along with our DID, DID document and proof, all of which are base64url encoded. The process is explained right below:

1) We verify the DID Document with the steps stated on the previous sub-chapter.

2) Finally, we verify the data’s signature signed with the DID Document’s JSON Web Key, utilizing the public key located there.

By making those checks we can verify the authenticity of data received. In a later chapter we showcase this process with an example.

3. Internet of Things Architecture

3.1. IoT Gateway

A gateway is a piece of networking hardware or software used in telecommunications networks that allows data to flow from one discrete network to another [4]. Gateways are distinct from routers or switches in that they communicate using more than one protocol to connect multiple networks and can operate at any of the seven layers of the Open Systems Interconnection model (OSI).

An Internet of Things (IoT) gateway provides a bridge (protocol converter) between IoT devices in the field, the cloud, and user equipment such as smartphones. The IoT gateway provides a communication link between the field and the cloud and may provide offline services and real-time control of devices in the field.

Interconnected devices communicate using lightweight protocols that do not require extensive CPU resources. To achieve sustainable interoperability on the IoT ecosystem, the dominant architectures for data exchange protocols are MQTT and CoAP. Programming languages such as C, Java, Python are the preferred choices for IoT application developers.

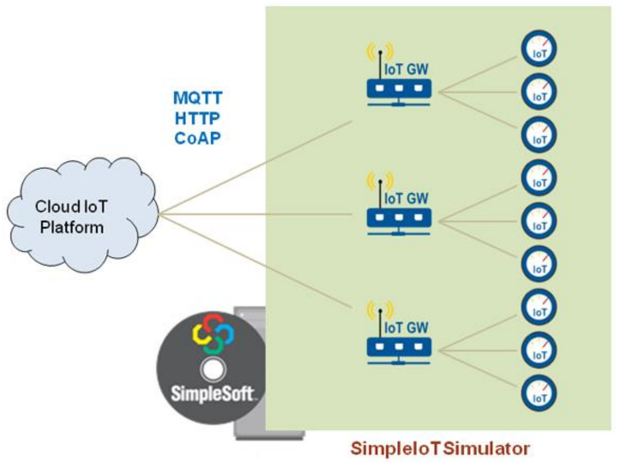


Figure 4: An example of IoT an Gateway connecting IoT devices and sensors to cloud-based computing and data processing [5]

3.2. The Constrained Application Protocol (CoAP)

3.2.1. Specifications and Features

The Constrained Application Protocol (CoAP) is a specialized Internet application protocol for constrained devices, as defined in RFC 7252 [6][7]. It enables constrained devices called "nodes" to communicate with the wider Internet using messages similar to, but simpler than, those of HTTP. CoAP is designed for use between devices on the same constrained network (e.g., low-power, lossy networks), between devices and general nodes on the Internet, and between devices on different constrained networks, both joined by an internet. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation.

CoAP is a service layer protocol, intended for use in resource-constrained internet devices, such as wireless sensor network nodes. CoAP is designed to easily translate to HTTP for simplified integration with the web, achieving interworking between the two protocols, while also meeting specialized requirements such as multicast support, exceptionally low overhead, and simplicity. Those requirements are important for Internet of things (IoT) and machine-to-machine (M2M) communication, which tend to be deeply embedded and have much less memory and power resources than traditional internet devices. Therefore, efficiency is especially important.

CoAP utilizes UDP as the underlying network protocol, DTLS for security and uses the same methods as HTTP (Get/Put/Post/Delete).

3.2.2. CoAP Group Communication

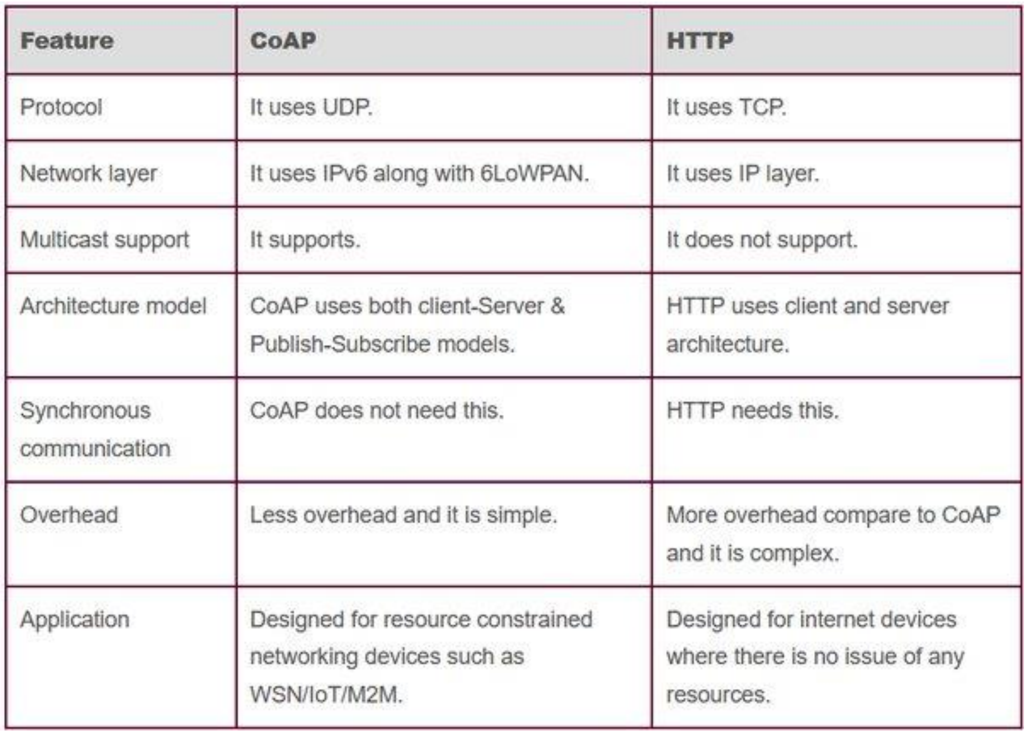
CoAP group communication plays a significant role in the IoT. In many CoAP application domains it is essential to have the ability to address several CoAP resources as a group, instead of addressing each resource individually (e.g., to turn on all the CoAP-enabled lights in a room with a single CoAP request triggered by toggling the light switch).

To address this need, the Internet Engineering Task Force (IETF) has developed an optional extension for CoAP in the form of an experimental RFC, Group Communication for CoAP - RFC 7390 [7][8]. This extension relies on **IP multicast** to deliver the CoAP request to all group members. Multicasting has certain benefits such as reducing the number of packets needed to deliver the request to the members. On the other hand, multicast also has its limitations, such as poor reliability and cache-unfriendliness.

An alternative method for CoAP group communication that uses **unicasts** instead of multicasts, relies on having an intermediary, such as a Gateway, where the groups are created. Clients send their group requests to the intermediary, which in turn sends individual unicast requests to the group members, collects the replies from them, and sends back an aggregated reply to the client. The intermediary thus acts as a concentrator.

3.2.3. HTTP and CoAP

CoAP is intentionally similar to HTTP but modified to work with constrained devices. The following figure compares CoAP and HTTP.



Feature	CoAP	HTTP
Protocol	It uses UDP.	It uses TCP.
Network layer	It uses IPv6 along with 6LoWPAN.	It uses IP layer.
Multicast support	It supports.	It does not support.
Architecture model	CoAP uses both client-Server & Publish-Subscribe models.	HTTP uses client and server architecture.
Synchronous communication	CoAP does not need this.	HTTP needs this.
Overhead	Less overhead and it is simple.	More overhead compare to CoAP and it is complex.
Application	Designed for resource constrained networking devices such as WSN/IoT/M2M.	Designed for internet devices where there is no issue of any resources.

Figure 5: Differences of CoAP and HTTP [9]

The classic architecture of a CoAP system consists of a CoAP client, a CoAP server, a REST CoAP proxy, and the REST Internet. Data are sent from CoAP clients, such as smartphones or any IoT device, to the CoAP server, and the same message is routed to REST CoAP proxy. The REST CoAP proxy interacts with devices outside the CoAP

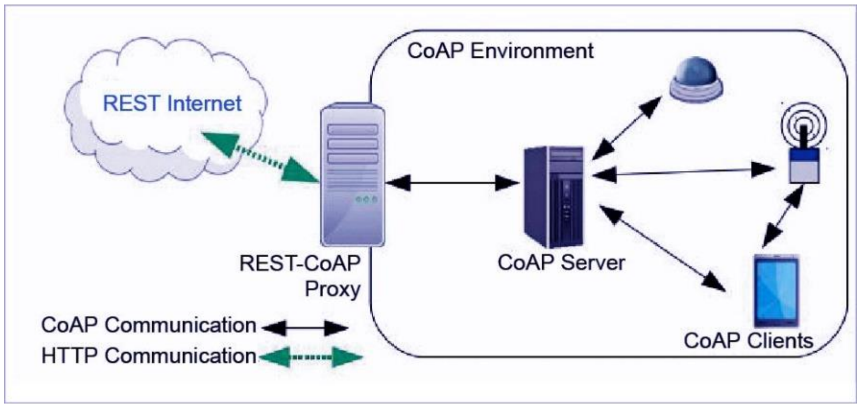


Figure 6: Architecture of CoAP Environment [10]

environment and uploads the data over the REST Internet, after translating the response from CoAP to HTTP. This is shown in the following figure.

3.3. IoT Device Setup

There are multiple ways to set up your IoT device and establish a relationship with the Gateway. The main options are:

1. The IoT device broadcasts a message informing of its installation to the network, including its IP address to the Gateway.
2. The Gateway can broadcast a message requesting the IoT devices to inform it of their availability and give it their IP addresses.
3. Directly submit the IP addresses of the devices to the Gateway. This is done with a request from a neutral resource, like a controller or client via a message. The IPs of the devices are not received from them directly.

After the Gateway receives information about the Devices, it adds them to an IP multicast address, or saves them to an array, so as to be able to later send CoAP requests to them to fulfill a request it receives from the Internet.

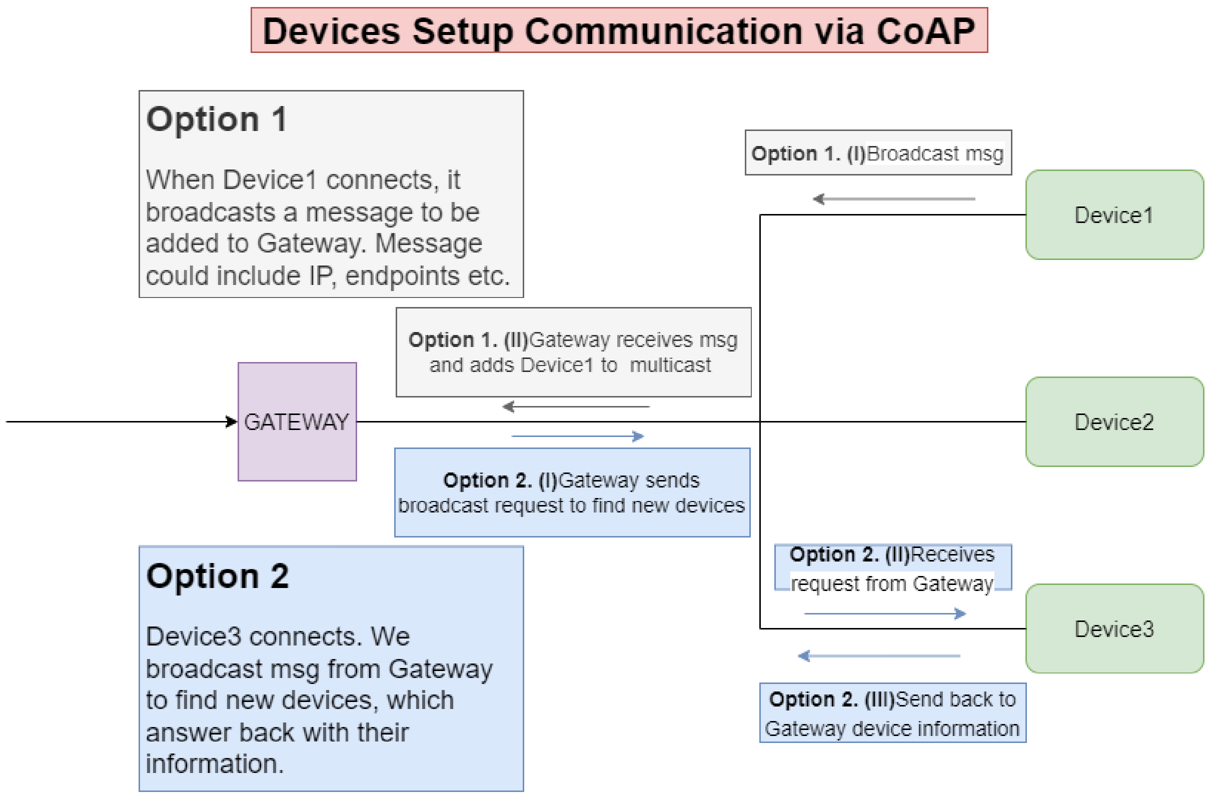

Devices Setup Communication via CoAP
Option 1
When Device1 connects, it broadcasts a message to be added to Gateway. Message could include IP, endpoints etc.
Option 1. (I)Broadcast msg
Device1
Option 1. (II)Gateway receives msg and adds Device1 to multicast
GATEWAY
Device2
Option 2. (I)Gateway sends broadcast request to find new devices
Option 2
Device3 connects. We broadcast msg from Gateway to find new devices, which answer back with their information.
Option 2. (II)Receives request from Gateway
Device3
Option 2. (III)Send back to Gateway device information

Figure 7: Option 1 & 2 illustration of Device Set Up Communication

3.4. IoT Device Communication

Here we provide an example of Group Communication, fulfilling a request received from the Internet, based on CoAP Group Communication.

After receiving a Request from the Internet, the gateway can carry it out with:

- A Multicast Communication
  - [1] Sends a **Multicast** *Request* to devices based on resource requested.
  - [2] Receives many *Responses* from devices.
  - [3] Gathers data into one Response, via aggregation or other method.
  - [4] Sends this Response to the entity that requested it, such as a web client, cloud, application.

- Many Unicast Communications
    1. [1] Sends many **Unicast** *Requests*, one for each device.
    2. [2] Receives many *Responses* from devices.
    3. [3] Gathers data into one Response, via aggregation or other method.
    4. [4] Sends this Response to the entity that requested it, such as a web client, cloud, application.

3.5. Decentralized Identifiers on IoT

Decentralized Identifiers (DIDs), received from the devices, can be used to:

- ✓ Find services the devices are offering.
- ✓ Authenticate the responses we receive.
- ✓ Secure the transfer of resources.

Specifically, did:self DIDs are created for each device, possibly by their owner. The corresponding DID Document needs to be altered only if its JSON Web Keys expire or if the services offered from the devices change. These DID documents can be directly transmitted to interested parties or stored in publicly accessible locations and can be verified as "correct" even if they are retrieved over an unsecured channel, as mentioned in a previous chapter, making them ideal for the IoT architecture.

4. The RIOT Operating System

4.1. Introduction

In our project we used the RIOT operating system to implement the did:self method in an actual IoT architecture. As we will explain in this chapter, the RIOT OS is a valuable tool for the Internet of Things. Fundamentally, an Operating System (OS) is characterized by a few key design aspects, such as the **kernel** and the **programming model** [11].

The kernel can either be

i. a monolithic program,
ii. follow a layered approach, or
iii. implement the microkernel architecture.

The programming model defines whether

i. all tasks are executed within the same context and have no segmentation of the memory address space, or
ii. each process can run in its own context and memory space.

The programming model is also linked to the available programming languages for application developers.

4.2. Description

RIOT is an **open-source microkernel**-based **operating system**, designed to match the requirements of **Internet of Things** (IoT) devices and other embedded devices [11][12]. It is a small operating system for **networked**, **memory-constrained systems** with a focus on **low-power wireless** Internet of Things (IoT) devices.

RIOT provides a microkernel, multiple network stacks, and utilities which include cryptographic libraries, data structures (bloom filters, hash tables, priority queues), a shell and more. A microkernel OS is structured as a tiny kernel that provides the minimal services used by a team of optional cooperating processes, which in turn utilize modules or libraries to provide higher-level OS functionality. **The microkernel itself lacks**

filesystems and many other services normally expected of an OS. Those services are provided by optional processes.

The real goal in designing a microkernel OS is not simply to “make it small.”. Since all the services, like device drivers, memory management, communication, file management and others, must be built in user space, the microkernel will provide a layer where a minimum set of services reside. So, **the microkernel architecture provides an abstraction for the hardware layer, so that it can be adapted to different hardware architectures, without changing the services offered to applications.**

Compared to other IoT operating systems, such as Contiki OS and Tiny OS, which are event-based and provide application development tools supporting a subset of the C language (contiki) and nestc (tinyos), RIOT fares better when it comes to **memory usage** and **support**.

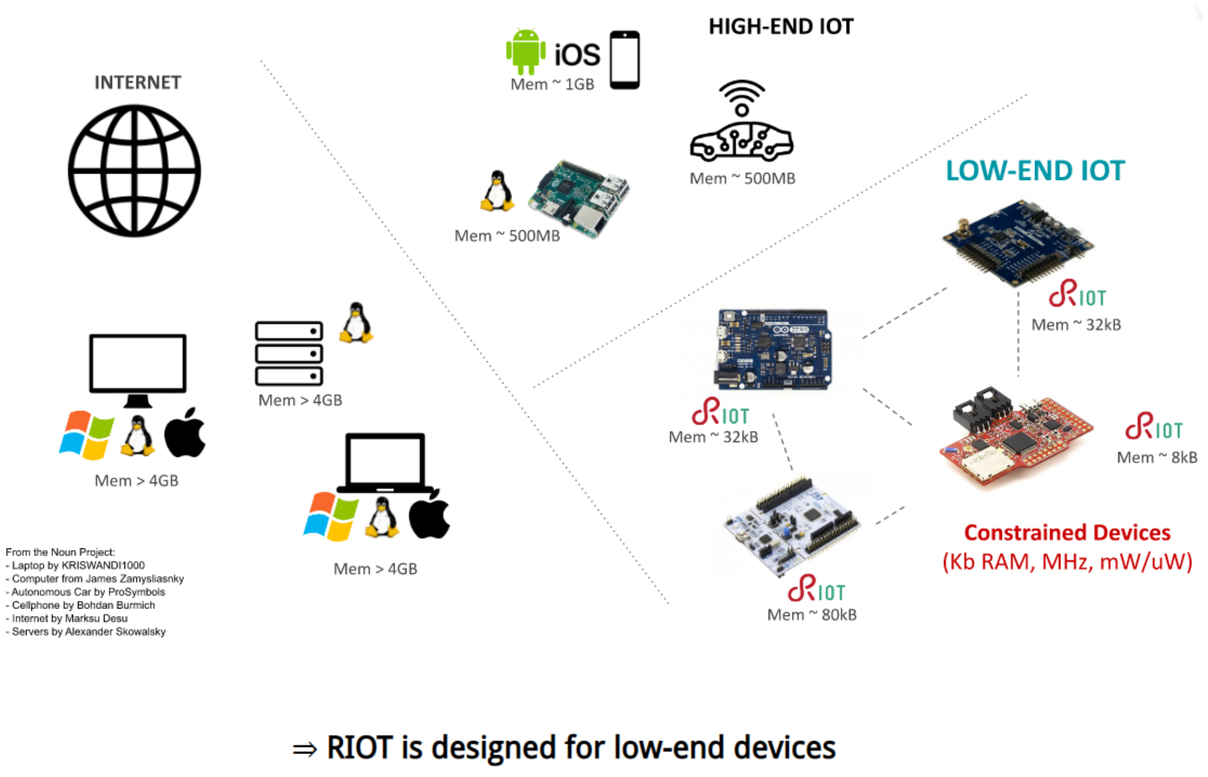

HIGH-END IOT
iOS
Mem ~ 1GB
INTERNET
Mem ~ 500MB
Mem ~ 500MB
LOW-END IOT
RIOT
Mem ~ 32kB
RIOT
Mem ~ 32kB
RIOT
Mem ~ 8kB
Mem > 4GB
Mem > 4GB
Mem > 4GB
Constrained Devices
(Kb RAM, MHz, mW/uW)
RIOT
Mem ~ 80kB
From the Noun Project:
- Laptop by KRISWANDI1000
- Computer from James Zamysliasnky
- Autonomous Car by ProSymbols
- Cellphone by Bohdan Burmich
- Internet by Marksu Desu
- Servers by Alexander Skowalsky
⇒ RIOT is designed for low-end devices

Figure 8: RIOT in the IoT realm [13]

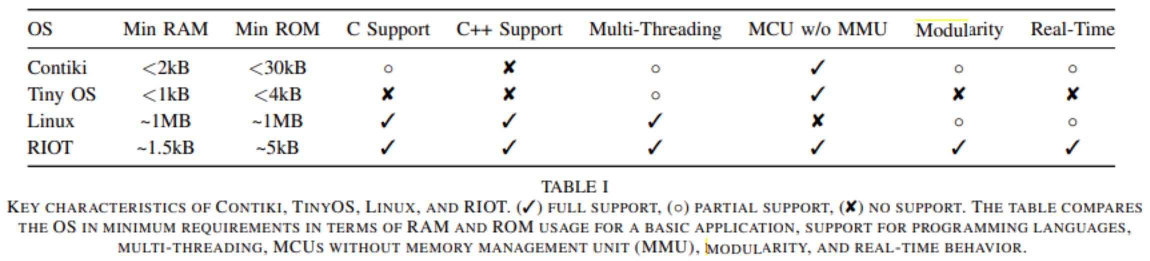
4.3. Features

In this section we mention a few features of RIOT OS:

- RIOT offers a “traditional” threading and scheduling scheme, POSIX-compliance, thus supporting **multi-threading** with standard API.
- RIOT offers **C** and (currently, partial) **C++ language** support, enabling powerful libraries.
- RIOT provides a TCP/IP **network stack** (IP oriented stacks and High-level network protocols: CoAP, MQTT-SN, etc.)
- RIOT can be extended with **external packages**; thus, features are provided as **modules.** The **modular microkernel** structure of RIOT makes it robust against bugs in single components.
- RIOT is based on **design objectives** including **energy-efficiency**, **small memory footprint**, **modularity**, and **uniform API access**, independent of the underlying hardware.

4.4. Comparison of IoT Operating Systems

As we can see from the image below and based on the features we mentioned at the previous sub-chapter, the RIOT OS surpasses other IoT Operating Systems, integrating successfully the IoT design objectives. Consequently, it is ideal for developing concepts on the Internet of Things, such as the did:self method.



OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	<2kB	<30kB	○	✘	○	✓	○	○
Tiny OS	<1kB	<4kB	✘	✘	○	✓	✘	✘
Linux	~1MB	~1MB	✓	✓	✓	✘	○	○
RIOT	~1.5kB	~5kB	✓	✓	✓	✓	✓	✓

TABLE I

KEY CHARACTERISTICS OF CONTIKI, TINYOS, LINUX, AND RIOT. (✓) FULL SUPPORT, (○) PARTIAL SUPPORT, (✘) NO SUPPORT. THE TABLE COMPARES THE OS IN MINIMUM REQUIREMENTS IN TERMS OF RAM AND ROM USAGE FOR A BASIC APPLICATION, SUPPORT FOR PROGRAMMING LANGUAGES, MULTI-THREADING, MCUS WITHOUT MEMORY MANAGEMENT UNIT (MMU), MODULARITY, AND REAL-TIME BEHAVIOR.

Figure 9: Key Characteristics of CONTIKI, TINYOS, LINUX AND RIOT [11]

5. Our Implementation

5.1. Objective

The goal of our thesis is to implement decentralized identifiers on IoT devices, specifically on RIOT OS, and examine if the DIDs created with did:self method can be used for security purposes. In this chapter, we explain our setup and implementation, while in the next chapter, we show that we can receive a DID from a device, as well as the DID Document and Proof, and we can authenticate the data sent from each device using the verified DID Document.

Our code implementation is saved on a GitHub repository located at https://github.com/KonstBet/did-self-riot-os, which includes the CoAP server running on IoT devices, written for the RIOT OS in C programming language, and the proxy Gateway, written in python, which are utilized as server and client, with the CoAP protocol, to fulfill requests from a client to multiple IoT devices.

5.2. Client & Gateway

We used the coap-client tool at the ubuntu terminal to create CoAP requests. Those requests are made to the Gateway or Devices according to our debugging needs. Coap-client is also used to display our examples in the next chapter.

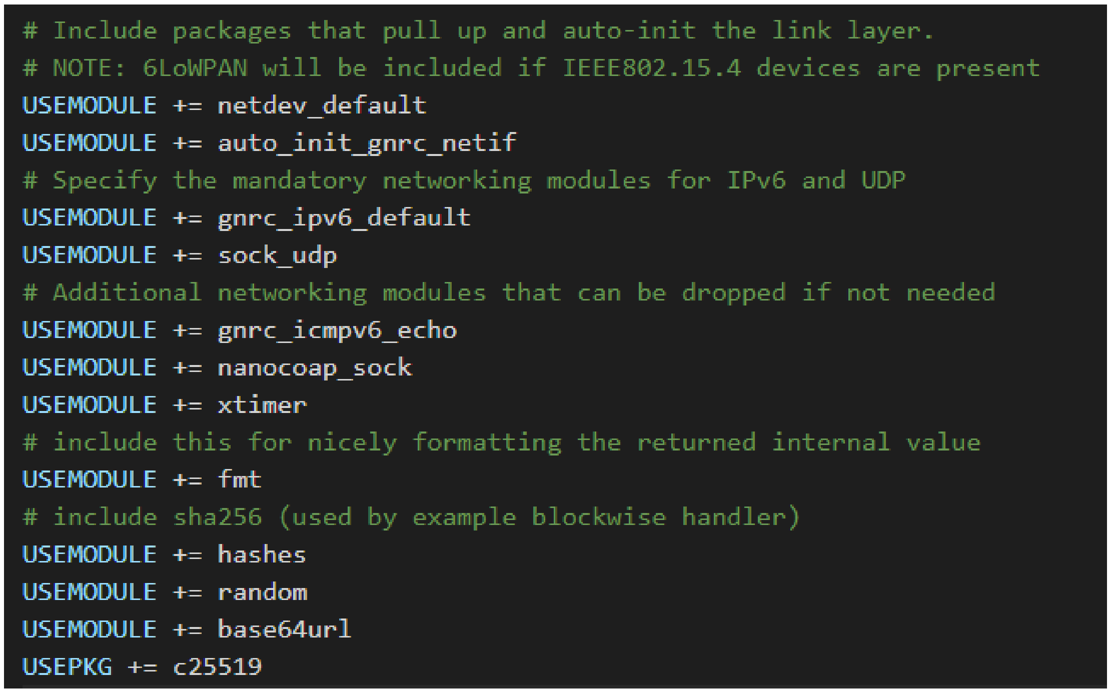
The Gateway is being utilized as a proxy. RIOT does not support yet the use of multicast in CoAP successfully. We added ourselves the devices' ipv6 addresses, by making a request to add a new device. Then, the gateway can make multiple unicast requests to each device, to fulfill a request sent to a group. **The Gateway successfully verifies the DID Document and Data we receive from each device**. Most important though, and the subject of this paper, is the implementation of DIDs with the did:self method on the Internet of Things, which is explained in the next sub-chapter.

5.3. IoT Device

As we mentioned in the previous chapter**, we are using the RIOT OS to develop our application.** The testing for the DID implementation of did:self method, **which we wrote in C programming language**, was done on the Windows Subsystem for Linux (WSL).

5.3.1. Modules

We included various modules needed for implementing the did:self method and running a CoAP server, as we can see in the image below, from our Makefile.



```
# Include packages that pull up and auto-init the link layer.
# NOTE: 6LoWPAN will be included if IEEE802.15.4 devices are present
USEMODULE += netdev_default
USEMODULE += auto_init_gnrc_netif
# Specify the mandatory networking modules for IPv6 and UDP
USEMODULE += gnrc_ipv6_default
USEMODULE += sock_udp
# Additional networking modules that can be dropped if not needed
USEMODULE += gnrc_icmpv6_echo
USEMODULE += nanocoap_sock
USEMODULE += xtimer
# include this for nicely formatting the returned internal value
USEMODULE += fmt
# include sha256 (used by example blockwise handler)
USEMODULE += hashes
USEMODULE += random
USEMODULE += base64url
USEPKG += c25519
```

Figure 10: Modules included in our project

Specifically, the first ones are used for running the CoAP server and the last four for the implementation of the DIDs with the did:self method.

- random module: Generation of random SSH private and public keys, which are used for DID Document and Proof verification.
- hashes module: Utilization of sha256 hash algorithm.
- base64url module: Encode data in string form. Both sha256 and base64url are used for verification purposes and secure network transfer.

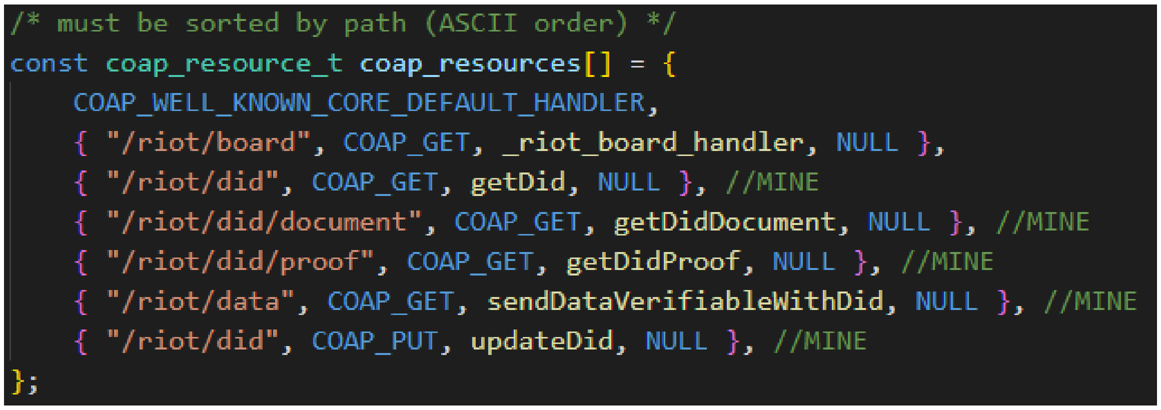
- c25519 module: In cryptography, Curve25519 is an elliptic curve used in elliptic-curve cryptography. More accurately, we used Ed25519, which is a public-key signature system.

5.3.2. Functions & Resources

The code was written in the C programming language. We created structures used for saving our DID Document and DID Proof variables. Utilizing the modules we stated above, we created functions for:

- The generation of random ed25519 public/private key pairs, used in DID Document and Proof.
- The signing of data with those SSH keys.
- The creation and encoding of variables included in the DID Document and Proof.
- The creation of the DID itself, the DID Document and the DID Proof, making use of the functions above, based on the did:self method.

The resources we are offering via the CoAP server, running on each device, can be seen in the following screenshot.



```c
/* must be sorted by path (ASCII order) */
const coap_resource_t coap_resources[] = {
    COAP_WELL_KNOWN_CORE_DEFAULT_HANDLER,
    { "/riot/board", COAP_GET, _riot_board_handler, NULL },
    { "/riot/did", COAP_GET, getDid, NULL }, //MINE
    { "/riot/did/document", COAP_GET, getDidDocument, NULL }, //MINE
    { "/riot/did/proof", COAP_GET, getDidProof, NULL }, //MINE
    { "/riot/data", COAP_GET, sendDataVerifiableWithDid, NULL }, //MINE
    { "/riot/did", COAP_PUT, updateDid, NULL }, //MINE
};
```

Figure 11: Resources our coap server is offering

Those, based on the request method are:

- ❖ GET:
  - ▪ /riot/board: get board name (basic function).
  - ▪ /riot/did: get all information about did, including document and proof.

  - /riot/did/document: get DID document.
  - /riot/did/proof: get DID id proof.
  - /riot/data: get data signed with the key in the DID Document.

- PUT:
  - /riot/did: update the DID.

The creation of the DID happens when we request it for the first time. The same DID is sent every time, until we update it with the PUT request.

5.4. Limitations & Challenges

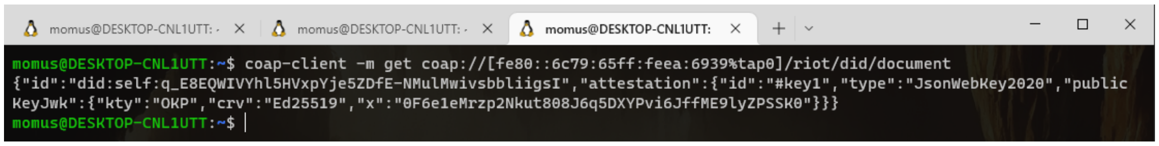
During the implementation we faced certain challenges and hit certain barriers. We state them below.

1. We could not utilize multicast addresses. As explained above, this is a RIOT limitation, which we bypassed by translating requests to a group to multiple unicast requests.
2. There are no preexisting libraries for JSON Web Token handling, so we implemented JWT in the C programming language, based on their formal documentation.
3. The application requires at least 218KB RAM on our microcontrollers.

6. Results

In this chapter we show the responses received from the device via the coap-client. Verifications of the DID Document and any data we receive from the devices is done successfully at our implementation of a proxy Gateway, which verifies that the did:self method was implemented successfully on RIOT OS.

The real responses are without whitespace as we see below.

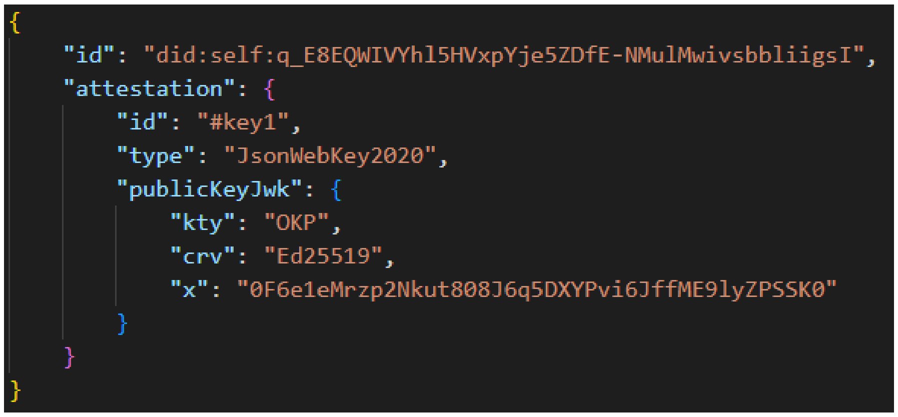


```
momus@DESKTOP-CNL1UTT:~$ coap-client -m get coap://[fe80::6c79:65ff:feea:6939%tap0]/riot/did/document
{"id":"did:self:q_E8EQWIVYhl5HVxpYje5ZDfE-NMulMwivsbbliigsI","attestation":{"id":"#key1","type":"JsonWebKey2020","public
KeyJwk":{"kty":"OKP","crv":"Ed25519","x":"0F6e1eMrzp2Nkut808J6q5DXYPvi6JffME9lyZPSSK0"}}}
momus@DESKTOP-CNL1UTT:~$
```

Figure 12: Get DID Document (No Whitespaces)

But for the purpose of showcasing our scheme, we will format the results. The first two resources are utilized only for debugging purposes. On production they should be removed.

- **GET DID DOCUMENT (JSON format – for debugging)**

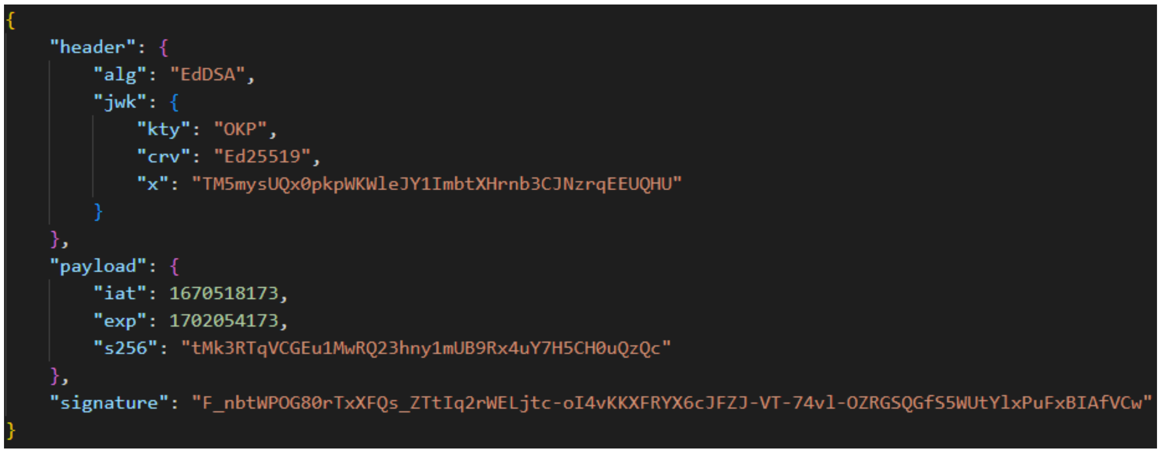


```
{
    "id": "did:self:q_E8EQWIVYhl5HVxpYje5ZDfE-NMulMwivsbbliigsI",
    "attestation": {
        "id": "#key1",
        "type": "JsonWebKey2020",
        "publicKeyJwk": {
            "kty": "OKP",
            "crv": "Ed25519",
            "x": "0F6e1eMrzp2Nkut808J6q5DXYPvi6JffME9lyZPSSK0"
        }
    }
}
```

Figure 13: DID Document (JSON)

Successfully returns the DID Document of the device. We need the DID Proof to authenticate it.

- **GET DID PROOF (JSON format – for debugging)**



```
{
    "header": {
        "alg": "EdDSA",
        "jwk": {
            "kty": "OKP",
            "crv": "Ed25519",
            "x": "TM5mysUQx0pkpWKWleJY1ImbtXHrnb3CJNzrqEEUQHU"
        }
    },
    "payload": {
        "iat": 1670518173,
        "exp": 1702054173,
        "s256": "tMk3RTqVCGEu1MwRQ23hny1mUB9Rx4uY7H5CH0uQzQc"
    },
    "signature": "F_nbtWPOG80rTxXFQs_ZTtIq2rWELjtc-oI4vKKXFRYX6cJFZJ-VT-74vl-OZRGSQGfS5WUtYlxPuFxBIAfVCw"
}
```

Figure 14: DID Proof (JSON)

Signature of the proof, signed to the base64url encoding of the header and payload separated by a dot, is needed for DID verification, as explained previously.

- **PUT DID (UPDATE)**

The update request returns the message “DID Updated” signaling the refresh of the DID. If we request the decentralized identifier afterwards, we can observe that it has been altered.

- **GET DID (Document and Proof – base64url encoded)**

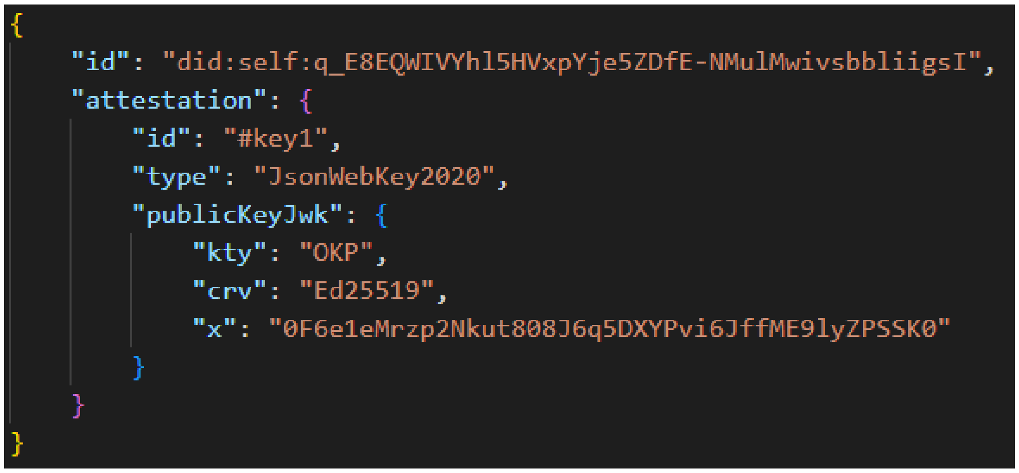
This request is the most important in our implementation. It returns the DID data we need to verify the Decentralized Identifier Document according to Chapter 2.2. Below, we have the response, which contains the information encoded in base64url. The RIOT OS has not implemented the addition of header parameters; therefore, we added the proof to the payload. The DID Document and DID Proof are separated by a space. Specifically, we have painted the Document blue, the Proof green, and the dots red.

eyJpZCI6ImRpZDpzZWxmOnFfRThFUVdJVllobDVIVnhwWWplNVpEZkUtTk11bE1
3aXZzYmJsaWlnc0kiLCJhdHRlc3RhdGlvbiI6eyJpZCI6IiNrZXkxIiwidHlwZSI6Ikpzb25
XZWJLZXkyMDIwIiwicHVibGljS2V5SndrIjp7Imt0eSI6Ik9LUCIsImNydiI6IkVkMjU1
MTkiLCJ4IjoiMEY2ZTFlTXJ6cDJOa3V0ODA4SjZxNURYWVB2aTZKZmZNRTlseV
pQU1NLMCJ9fX0
eyJhbGciOiJFZERTQSIsImp3ayI6eyJrdHkiOiJPS1AiLCJjcnYiOiJFZDI1NTE5IiwieCI6
IjlwMEFhRDgwZUEzbHhOYzlROUJWWkx2WjJObVdBWVZzX0s3Wk1VbVVwSXci
fX0.eyJpYXQiOjE2NzE3Mjk1MjYsImV4cCI6MTcwMzI2NTUyNiwiczI1NiI6ImI2X2l
QX0h6cGJTZ3BXbURCOVdWMjhtZ3ZIVXhtNzg4UVk2VHZ1blFNWEUifQ.cuDvpP
eSWjtF5JnL4nXSvfGOR8v6xb-E-
p235g8NZffxsY6eM_E6YJ8j5etz2lY4RQxvyZUaUVNNo7m049f6Dw

As we can see, all the information is base64url encoded based on the did:self method and to decrease payload size. Below, we will analyze each part of the response, giving the result after base64url decode. Segments are divided by the dots.

- DID Document (Blue segment)

  This segment is the base64url encoded string of the DID Document. If we decode it, we receive the JSON we see below.



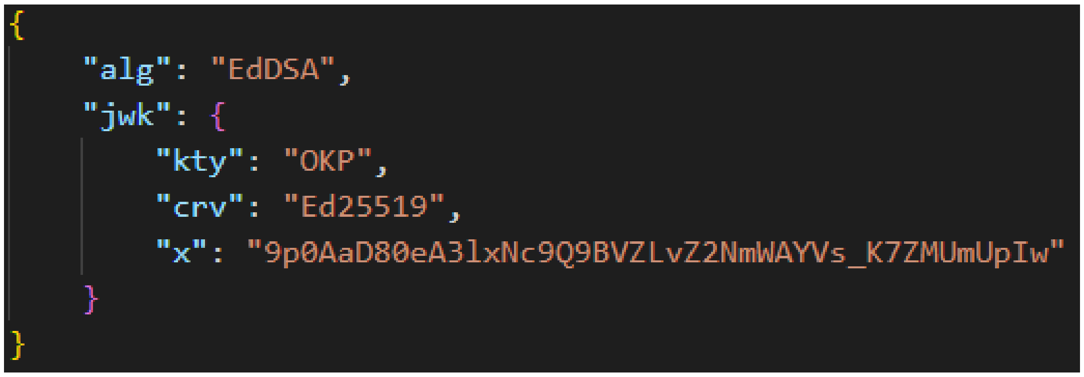
```
{
    "id": "did:self:q_E8EQWIVYhl5HVxpYje5ZDfE-NMulMwivsbbliigsI",
    "attestation": {
        "id": "#key1",
        "type": "JsonWebKey2020",
        "publicKeyJwk": {
            "kty": "OKP",
            "crv": "Ed25519",
            "x": "0F6e1eMrzp2Nkut808J6q5DXYPvi6JffME9lyZPSSK0"
        }
    }
}
```

Figure 15: DID Document (after decoding)

- <u>DID Proof (Green Segment)</u>

  The DID Proof has the design of a JSON Web Token.

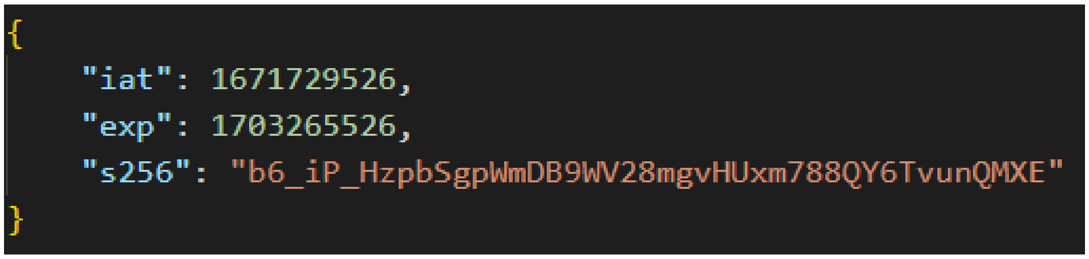
  The first segment includes the header.



```
{
    "alg": "EdDSA",
    "jwk": {
        "kty": "OKP",
        "crv": "Ed25519",
        "x": "9p0AaD80eA3lxNc9Q9BVZLvZ2NmWAYVs_K7ZMUmUpIw"
    }
}
```

Figure 16: DID Proof Header (after decoding)

  The second segment includes the payload.



```
{
    "iat": 1671729526,
    "exp": 1703265526,
    "s256": "b6_iP_HzpbSgpWmDB9WV28mgvHUxm788QY6TvunQMXE"
}
```

Figure 17: DID Proof Payload (after decoding)

  Finally, the third segment is the signature of the header and payload separated by a dot, by the header's JWT.

The signatures are created according to the formal documentation to the base64url encoded strings of each data and not to the JSON string.

In the end we can use this response to verify the Decentralized Identifier's Document. We successfully verify the DID Documents on our implementation of a proxy Gateway, written in Python, following the instructions provided in Chapter 2.2, verifying that our implementation on RIOT OS is successful.

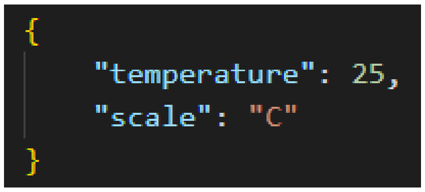
➢ **GET DATA (Verified by the included DID Document)**

The response includes the same information as GET DID, plus the data separated by a space. The data is represented as shown below.

eyJ0ZW1wZXJhdHVyZSI6MjUsInNjYWxlIjoiQyJ9.helJL0c1-_l4tfXL2nZoFEW8NFGSZ-nRRyUwhqgrG4WA31IQRFrSVSTMn20T9-f7Yk07mxyH85Q8Lsdp2Q9CCQ

First, we verify the DID Document using the Proof as explained in Chapter 2.2.

In our base64url encoded string of the data, we have two segments separated by a dot. The first segment is the data we wish to transfer. In this case it is the JSON shown next, after base64url decode.



```
{
    "temperature": 25,
    "scale": "C"
}
```

Figure 18: Data received, verifiable by DID Document

The second segment is the signature of the base64url encoded string of our data, signed with the JWT created for the DID Document and verified to the client by the public key given in the DID Document.

Finally, we observe that we can authenticate our data successfully using Decentralized Identifiers, as proven from our implementation of the python Gateway, which authenticates the data successfully, following the instructions in Chapter 2.3.

7. Conclusion & Future Work

This paper shows that we can use Decentralized Identifiers for the Internet of Things. Particularly, we can utilize the did:self method successfully on the RIOT Operating System, to fulfill our security needs and authenticate any data we receive from our devices. Future work is required to complete the IoT environment of our paper.

The successful use of multicasting for communicating with a group of devices is an important part of the Internet of Things. Group communication with the verification of decentralized identifiers is a good step on completing the architecture.

Research on the use of a HTTP-CoAP Gateway which is able to use multicast for specific groups of devices and verify the content with the did:self method, specifically, can be conducted.

The CoAP server module on RIOT OS requires a lot of memory on the device, for the IoT realm, and research on using a network stack that requires the minimum system memory is essential.

Finally, we could study diverse ways of using the Decentralized Identifier for security. Other forms of authentication, using the DID Document created from the method did:self, are also possible, and worth further research.

8. References

[1] W3C, "Decentralized Identifiers (DIDs) v1.0". (2022), Available at: https://www.w3.org/TR/did-core/.

[2] W3C, "DID Formal Objection FAQ". (2021), Available at: https://www.w3.org/2019/did-wg/faqs/2021-formal-objections/

[3] Nikos Fotiou, Vasilios A. Siris, George Xylomenos and George C. Polyzos, "IoT Group Membership Management Using Decentralized Identifiers and Verifiable Credentials", Future Internet (2022), Available at: https://www.mdpi.com/1999-5903/14/6/173

[4] Wikipedia, "Gateway (telecommunications)". (2022), Available at: https://en.wikipedia.org/wiki/Gateway_(telecommunications)

[5] SimpleSoft, "SimpleIoTSimulator", (2022), Available at: https://www.simplesoft.com/SimpleIoTSimulator.html

[6] IETF, RFC, "The Constrained Application Protocol (CoAP)". (2014), Available at: https://www.rfc-editor.org/rfc/rfc7252

[7] Wikipedia, "Constrained Application Protocol". (2022), Available at: https://en.wikipedia.org/wiki/Constrained_Application_Protocol

[8] IETF, RFC, "Group Communication for the Constrained Application Protocol (CoAP)". (2014), Available at: https://www.rfc-editor.org/rfc/rfc7390

[9] RF Wireless World, "CoAP vs HTTP". Available at: https://www.rfwireless-world.com/Terminology/Difference-between-CoAP-and-HTTP.html

[10] Javatpoint, "IoT Session Layer Protocols", Available at: https://www.javatpoint.com/iot-session-layer-protocols.

[11] Emmanuel Baccelli, Oliver Hahm et al, "RIOT OS: Towards an OS for the Internet of Things". (2013), Available at: https://hal.inria.fr/hal-00945122/document

[12] RIOT OS TEAM, "RIOT Documentation". (2022), Available at: https://doc.riot-os.org/

[13] RIOT OS, "Content of the course (Introduction)". See https://riot-os.github.io/riot-course/slides/01-introduction/#1