Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN

GRADO EN INGENIERÍA EN TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

Área de Ingeniería Telemática

Analisis de protocolos de bootstrapping para IoT

D. Julian Niklas Schimmelpfennig

TUTOR: D. Dan García Carrillo

TUTOR: D. Rafael Marín López

FECHA: Mayo de 2022

Resumen

En las últimas décadas, los datos digitales se han convertido en un bien valioso. El Internet de las Cosas (IoT según sus siglas en ingles) sigue esta tendencia con el objetivo principal de aprovechar la información recogida por los sensores. Estos miden la temperatura, detectan el humo o el movimiento y forman un dispositivo IoT junto con el microcontrolador conectado. Para procesar estos datos, hay que transmitirlos en una red informática a diferentes entidades. Pero, ¿cómo entran en la red estos dispositivos IoT, algunos de ellos con capacidades muy limitadas? ¿Cómo pueden los dispositivos autorizados comunicarse de forma segura en la red? Esto se hace en un proceso llamado *bootstrapping*.

Los dispositivos IoT con capacidades limitadas de diferentes fabricantes con diversas arquitecturas de microcontroladores se unen para medir datos del mundo físico, interactuar con el entorno o entre sí. Las diferencias en la potencia de cálculo disponible no podrían ser mayores, pero todos ellos deben utilizar algoritmos criptográfico para lograr la integridad, la autenticidad y la confidencialidad. Para generar el material criptográfica necesario durante el bootstrapping, se puede utilizar el Extensible Authentication Protocol (EAP) [RFC 3748] como framework de autenticación. Los métodos EAP, son un mecanismo de autenticación específico puede ser elegido para cada dispositivo individualmente para seleccionar un algoritmo compatible.

Los dispositivos IoT alimentados por batería deben utilizar protocolos adaptados para su uso en entornos restringidos para maximizar su tiempo de actividad. El Constrained Application Protocol (CoAP) [RFC 7252] fue diseñado como un protocolo de transferencia web especializado para redes con altas tasas de error de paquetes, velocidades de datos en el rango de uno a dos dígitos de kilobytes por segundo y nodos restringidos. Aunque CoAP hace uso de la arquitectura REpresentational State Transfer (REST) como lo hace HTTP, no es una versión comprimida o adaptada con menor sobrecarga, sino un protocolo diferente. Además, CoAP utiliza UDP como capa de transporte en lugar de TCP y, por

tanto, requiere implementar estas características para garantizar el orden en las capas superiores.

Hypermedia as the Engine of Application State (HATEOAS) es una característica de REST que puede utilizarse para lograr el objetivo de dirigir el proceso de autenticación. Dado que no se requiere ningún conocimiento adicional a la interfaz solicitada para el acceso y la navegación, la estructura hipermedia simplifica el acceso de los clientes a la aplicación.

El Internet-Draft *EAP-based Authentication Service for CoAP* (CoAP-EAP) encapsula los paquetes EAP en CoAP para permitir la generación de material criptográfico incluso con redes y nodos con capacidades limitadas. Se trata de un protocolo que puede asociarse a la fase *bootstrapping* en el ciclo de vida de dispositivos IoT. Entre otras cosas, el último Internet-Draft de la versión 06 de CoAP-EAP hace uso de HATEOAS para garantizar el ordenamiento de los paquetes deduciendo el estado actual del proceso de autenticación sólo en base al estado del hipermedio. Esto se hace creando un nuevo recurso en el dispositivo IoT para cada paso durante la autenticación EAP para indicar su estado actual.

Este trabajo proporcionará una implementación del Internet-Draft de CoAP-EAP en su última versión. La prueba de concepto está escrita en el lenguaje de programación C y utiliza el motor Erbium REST, que es una biblioteca oficial del sistema operativo de código abierto Contiki. Los microcontroladores se han emulado utilizando el simulador Cooja y se comunican con un controlador CoAP-EAP durante la autenticación.

Los recursos web en Erbium se definen estáticamente antes de iniciar un dispositivo IoT. Analizando la estructura de datos subyacente para los recursos en Erbium, encontramos una manera de lograr la creación dinámica de recursos web durante el tiempo de ejecución. El sistema operativo completo, incluido el código adaptado del motor REST, sólo consume unos pocos kilobytes de memoria en el dispositivo IoT y proporciona una de las primeras implementaciones CoAP-EAP compatibles con HATEOAS.

Abstract

Over the last decades, digital data has become a valuable commodity. The Internet of Things (IoT) follows this trend with its main goal to benefit from the gathered information by sensors. They can measure temperature, detect smoke or movement and form an IoT device together with the connected microcontroller. In order to further process this data, it has to be forwarded in a computer network to different entities. But how do these constrained devices enter the network? How can approved devices communicate securely in the network? This is done in a process called *bootstrapping*.

Constrained devices from different manufacturers with various microcontroller architectures come together to measure data from the physical world, interact with the environment or each other. Differences in the available computing power could not be greater but all of them should use cryptographic algorithms to achieve integrity, authenticity and confidentiality. In order to generate the needed cryptographic key material during bootstrapping, the Extensible Authentication Protocol (EAP) [RFC 3748] can be used as an authentication framework. EAP methods, a specific authentication mechanism can be chosen for each device individually to select a supported algorithm.

The battery-powered IoT devices should use protocols tailored for use on constrained environments to maximize their uptime. The Constrained Application Protocol (CoAP) [RFC 7252] was designed as a specialized web transfer protocol for networks with high packet error rates, data rates in the one- to two-digit kilobyte per second range and constrained nodes. Even though CoAP makes use of the REpresentational State Transfer (REST) architecture as HTTP does, it is not a compressed or adapted version with lower overhead but a different protocol. Furthermore, CoAP uses UDP as transport layer instead of TCP and therefore requires an implementation for ordering guarantee in the higher layers.

Hypermedia as the Engine of Application State (HATEOAS) is a characteristic of REST that can be used to accomplish the goal of steering the authentication process. The hypermedia structure simplifies clients' access to the application, because no additional knowledge of the requested interface is required for access and navigation.

The Internet-Draft *EAP-based Authentication Service for CoAP* (CoAP-EAP) encapsulates EAP packages in CoAP to allow the generation of key material even within constrained networks and with limited nodes. CoAP-EAP can be assigned to the *bootstrapping* phase in the IoT lifecycle. Among other things, the latest draft version 06 of CoAP-EAP makes use of HATEOAS to guarantee the ordering of packages by deducing the current state of the authentication process only based on the state of the hypermedia. This is done by creating a new resource on the IoT device for each step during the EAP authentication to indicate its current state.

This work will provide an implementation of the CoAP-EAP draft in its latest version. The proof-of-concept is written in the programming language C and uses the Erbium REST engine, which is an official library of the open-source operating system Contiki. Microcontrollers have been emulated using the Cooja Simulator and communicate with a CoAP-EAP controller during the authentication.

Web resources in Erbium are statically defined before starting an IoT device. By analyzing the underlying data structure for resources in Erbium, we found a way to achieve the dynamic creation of web resources during runtime. The full operating system including the tailored code of the REST engine only consumes a few kilobytes of memory on the IoT device and provides one of the first HATEOAS-compliant CoAP-EAP implementations.

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form at any other institution or university.

I officially ensure that this work has been written solely on my own. I herewith officially ensure that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I agree that the digital version will be used to subject the paper to plagiarism examination.

	1 8 . 5 . 2 0 2 2
Julian Niklas Schimmelpfennig	Date

Contents

List of Figures

1. Introduction and Motivation

The Internet of Things (IoT) is a concept in computer science in which information from the physical world are gathered and provided in a global computer network to make use of that gained information [1].

In industry and building automation, a sensor e.g. for the current temperature can provide its measured data to a connected microcontroller. In this example, the sensor connected to the microcontroller form the so-called IoT device. In the most basic way, it will process the data and forwards it over a computer network to entities, which then could cause a change on a second IoT device if the measured temperature is above or below a given threshold. This triggers a change in the percolation of an automated thermostatic valve in order to higher or lower the room temperature. The benefit of the gathered information *temperature* from the physical world would be a constant room temperature and savings in heating costs [2].

The term *IoT controller* refers in this work to an entity inside the network that authenticates IoT devices. It may process or forward delivered data from the IoT devices as well if it combines multiple entities in one place.

Inside the computer network the IoT device communicates not only to the IoT controller, but to other IoT devices for machine-to-machine communication and network services (e.g. to a time server) as well [3]. IoT devices, IoT controllers and other devices in the network should implement a basic set of security features. Especially the use of encryption is important to keep the information secret from all but authorized parties. Integrity and message authentication are needed to ensure, that exchanged data between two communication partners are not altered accidentally or manipulated during the transmission and its source is the actual sender that pretends to be it. Among other things, common practice encryption algorithms and authentication methods from x86

architecture cannot be used on IoT devices due to their low computing power and battery operation [4].

Furthermore, only trustworthy devices should be allowed to enter the network and communicate with other devices and services inside the network. The decision, if whether an IoT device is trustworthy or not and the authentication process itself are among others part of a phase called *bootstrapping* [5]. Bootstrapping is one phase that an IoT device passes during its lifecycle. Prior phases to bootstrapping are e.g. evaluating and assembling the hardware of the IoT device itself, deploying the operating system and the physical placement of a device. The process of bootstrapping can make use of different protocols and technologies. One of the goals of bootstrapping is to receive cryptographic key material which can be used to ensure the authenticity between the communication partners and establish an encrypted communication channel within the network, after joining the network. Post bootstrapping refers among others to the normal operation phase in the IoT lifecycle, where additional key material could be generated [6].

One protocol for bootstrapping IoT devices is the Extensible Authentication Protocol (EAP). It supports various authentication methods and mechanisms that can be used without having to commit to a specific method in advance. The IoT controller can use different procedures with each IoT device individually to generate cryptographic key material. This flexibility of EAP is a great advantage, especially in large institutions, where many devices from different manufacturers have to be authenticated in a short time. With the Constrained Application Protocol (CoAP), there is also a web transmission protocol that has a lower overhead compared to the Hypertext Transfer Protocol (HTTP) [7] and was specifically developed for the use on constrained devices [8]. The Internet-Draft *EAP-based Authentication Service for CoAP* (CoAP-EAP) uses CoAP as an EAP lower layer to authenticate constrained devices by encapsulating EAP packages in CoAP messages and benefits from CoAP's advantages over other application-layer protocols.

CoAP-EAP can be used for bootstrapping IoT devices and makes use of the HATEOAS principle. Therefore, web resources need to be created and deleted dynamically on the side of the IoT device to indicate the current state of the process. As a result of this

bootstrapping technology, a Master Session Key (MSK) is generated which builds the foundation for the upcoming secure communication in the network.

In this work, the first implementation of the latest version 06 of CoAP-EAP based on the *Erbium (Er) REST Engine* [9] will be published. Erbium is an official library for the IoT operating system *Contiki* [10] written in the programming language C and does not support HATEOAS by default. Therefore, Erbium had been analyzed in detail to understand the mechanism of creating web resources. The result of this work represents source code written in C which can be executed after its compilation on IoT devices to authenticate them by using CoAP-EAP. The source code contains the whole operating system Contiki including Erbium, extended with the HATEOAS functionality and adapted for the usage with CoAP-EAP.

Structure of this Work

In the background chapter 2, the fundamental protocols and paradigms are introduced which are necessary to understand the objective in detail. It will outline the need for IoT-specific protocols and what the term bootstrapping means. It also introduces the HATEOAS principle and illustrates the steps for establishing a secure connection between the IoT device and IoT controller using CoAP-EAP.

Time management refers to finishing this scientific project as efficiently as possible in terms of chronological order and energy within a certain time limit. Therefore, a time schedule has been created to verify that the goals set for this project fit within the time frame and what sequence will allow for an optimal workflow. The schedule can be found in appendix A.1.

Chapter 3 will make a transition between the state of the art as described in chapter 2 and clarifies the objectives of this work. The implementation chapter 4 will target the technical details of the Erbium REST engine and its management of web resources, to implement the HATEOAS principle in a proof-of-concept using CoAP-EAP.

Finally, the results will be presented and discussed.

2. Background

This chapter introduces terminology and the technical background so that the following chapters become understandable. It starts with an introduction to basic security services and characteristic features of IoT devices when comparing them to the x86 architecture. After that, the IoT device lifecycle is introduced, followed by a structured explanation of the involved protocols and standards.

2.1. Security Services

Cryptographer Christoph Paar (Ruhr-University Bochum, Germany) defines the most relevant security services in *Understanding Cryptography* [11] as follows:

1. **Confidentiality**: Information is kept secret from all but authorized parties.
2. **Integrity**: Messages have not been modified in transit.
3. **Message Authentication**: The sender of a message is authentic.

Exchanged messages between the IoT device and the IoT controller and generally in the network should not be able to be changed undetected during its transmission in the computer network (violation of integrity and authenticity) or read by an unauthorized party (violation of confidentiality) [12]. The encryption and decryption of messages are therefore ideally performed directly in the sender and receiver themselves. For the use of such end-to-end encryption, a prior key agreement is necessary, with which the messages are encrypted by the sender and decrypted by the recipient [13].

2.2. IoT Characteristic Features

Methods for negotiating a secret key for the use of encryption between sender and receiver by using an insecure communication channel exist since 1978 with the Diffie-Hellman key exchange (DHKE) [14]. Strong symmetric encryption algorithms such as the Advanced

Encryption Standard (AES) have also been around since 2004 [15]. Today's x86 desktop computers and even smartphones nowadays can compute cryptographic operations [16]. They are accelerated by outsourcing certain operations into hardware [17] and one may ask: Why are new algorithms and protocols needed?

Computers require a constant power supply during operation. This is provided in smartphones and notebooks via rechargeable batteries. In desktop computers, it is provided by the power supply unit. IoT devices are battery-powered and do not use x86 processor architecture but microcontrollers with a fraction of its memory and computing power [4]. In particular, cryptographic operations cause additional computing time on the processors, which immensely reduces the lifetime of a device. The goal is to maximize the uptime of an IoT device without reducing security objectives. Thus, new resource-saving protocols are being developed specifically for microcontrollers [18].

2.3. IoT Device Lifecycle

Figure 2.1 shows the Generic lifecycle for IoT service. In order to achieve the mentioned security goals in section 2.1, we have to make sure, that both the IoT device and the IoT controller are authentic and those they pretend to be. Especially the verification, if whether an IoT device is allowed to join a security domain or not, is a challenge due to the big amount of devices that should be used and deployed at the same time. The security domain is a network where only authorized and therefore bootstrapped IoT devices can communicate.

When deploying IoT devices such as light sensors or smoke detectors in new buildings, their physical installation at the right location needs to be scheduled early during the planning phase. Structured cabling for IT infrastructure in the building e.g. for industrial wireless access points with IoT connectivity depends on the IoT device's location used in the future to ensure that signal strength is sufficient. In this process, a lot of different companies are involved and the commissioning of the IT infrastructure may start years later but has to be taken into account at an early stage [19].

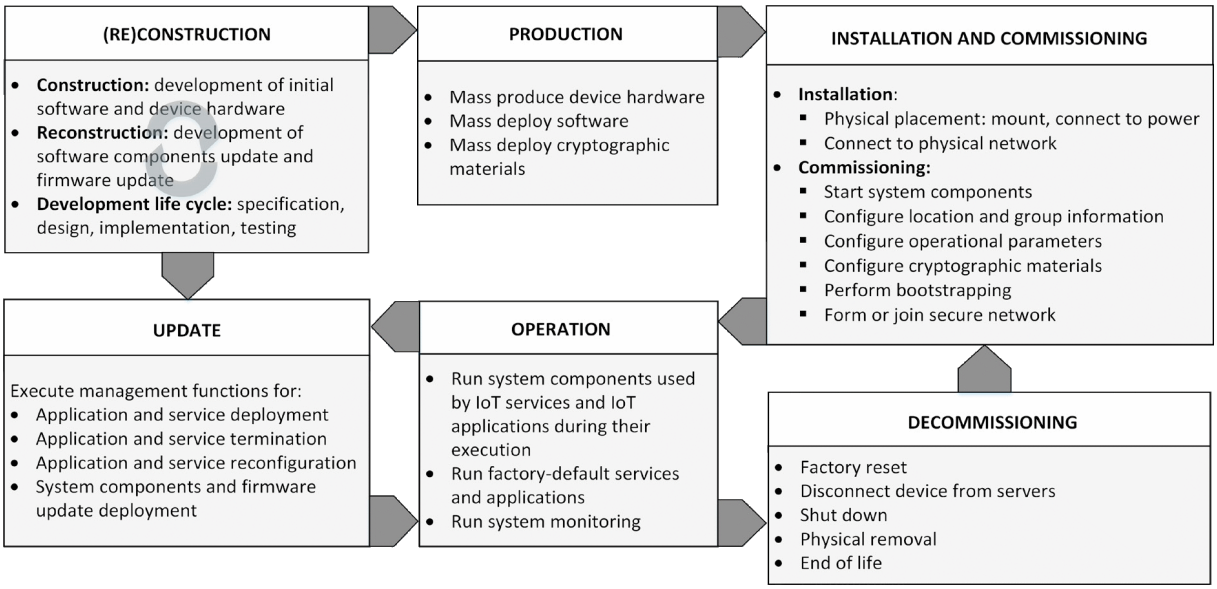


Figure 2.1 - Generic lifecycle for IoT service - Image taken from [6]

2.3.1. The Bootstrapping Phase

The bootstrapping process itself is one part of the installation and commissioning phase in the lifecycle of an IoT device. In the previous phases, the microcontrollers layout has to be designed and manufactured and the needed software components (operating system and third-party applications) need to be developed and tested. After those phases of construction and production, the device will be placed in its operating location: the bootstrapping process begins [6].

The process of getting a device fully operational, including the establishment of secure communication channels inside the network by using generated cryptographic key material, is called *bootstrapping*. Bootstrapping includes multiple sub-processes such as authentication, authorization and key distribution to enroll trustworthy IoT devices members of the security domain. In the generic lifecycle, it can be assigned to the commissioning stage [5].

Because IoT devices are developed and designed for particular tasks and use cases, IoT devices in larger domains are likely manufactured by different vendors. This

requires interoperability between the different manufacturer ecosystems and a uniform bootstrapping process by using standardized protocols [12].

The bootstrapping process finishes with generating the necessary key material, which can be used to establish post-bootstrapping security associations. The IoT controller also allows the IoT device to communicate with other entities inside the network such as time or key distribution servers or even Internet access [20].

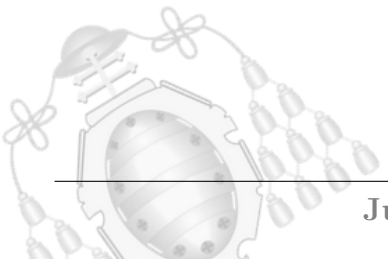
2.3.2. Post-Bootstrapping

The IoT controller might not be the entity granting access to the Internet. Inside the network can exist multiple entities for various services. There can be one entity to distribute the actual time to clients and a separate entity that grants internet access. The communication with these entities inside the network needs to be secured. These new capabilities can be used because bootstrapping was performed. The IoT device is a member of the security domain and gets access rights from the IoT controller.

From this point on, the generated key material for the security association protocols such as DTLS [21] and OSCORE [22] is derived from the negotiated Master Secret Key (MSK) at the beginning of this phase.

Post-bootstrapping affects more than only enabling a secure channel. Also, in cases of replacements of outdated IoT devices or disabling devices that are vulnerable to unresolved zero-day exploits, decommissioning is an important part of the IoT device lifecycle [23]. Decommissioning requires an unrecoverable removal of any cryptographic key material stored on the device and a factory reset to not leak information about the insides of the IT infrastructure before its disposal [6].

Among challenges in bootstrapping, the costs of the post-bootstrapping phase in the IoT device lifecycle (operating, monitoring, updating and decommissioning) should not be underestimated [12].



2.4. Protocols and Paradigms

After introducing IoT characteristics, the related protocols and paradigms are explained. It should provide the reader with an understanding, of why the development of IoT-specific application layer protocols was necessary and which components from the pre-IoT era can be further used.

2.4.1. Representational State Transfer (REST)

Since the beginning of the 2000s, Representational State Transfer (REST) has been an important principle for standardizing interfaces for web services. Roy Thomas Fielding introduced this architectural style in his dissertation [24].

A stateless client-server protocol is used to implement the REST paradigm. This can be HTTP or in the context of IoT devices CoAP (see 2.4.3) as an application layer protocol.

HTTP specifies request methods. The HTTP GET method requests information from a specified resource on a webserver. GET requests must be *idempotent* and *safe*. Idempotent means that sending the same request multiple times has no different effect than sending it once. *Safe* implies that this method only retrieves information and does not cause any other effects [25]. If a GET request causes e.g. the creation of a new resource (POST requests are used for that) or is neither idempotent nor safe, the server-side implementation is not REST compliant, because the server's behavior is different from what is expected.

All information that is needed to recover the page's state needs to be included in the request. The Unique Resource Identifier (URI) identifies only the resource, while the HTTP header may contain information such as access type (GET, PUT, POST, ...), return format or authentication.

URIs must be unambiguous, permanently identifiable and accessible via standard methods of the application protocol (HTTP or CoAP requests). They only specify the location, but not the content that the web service provides. Due to the statelessness and assuming prior

authentication, all necessary data to answer a request is delivered only depending on the information received in the request. This simplifies machine-to-machine communication (e.g. HTML and XML parser) and makes communication less error-prone [24].

2.4.2. Hypermedia as the Engine of Application State (HATEOAS)

HATEOAS is a limitation of previously explained REST application architecture. With HATEOAS, a client interacts with a server application that dynamically provides information via hypermedia. A client requires no or minimal existing knowledge of how to interact with a REST-conform application [26]. He only needs to know the URI. An intuitive example from the "The RESTful cookbook" [27] demonstrates the wanted behavior:

A customer is logged in to his online banking account. He does not have any money in his account because his last transfer or payment put it in the red. In that case, he should not be allowed to get his balance even more negative. The only option he should have from now on is to deposit more money until it is again in the black.

This behavior can be achieved even though the customer uses the same resource `GET /account/12345 HTTP/1.1` in online banking. The response to this GET request would normally return more options by providing links where he can do the specified action like transferring money. But the available options, delivered from the server in the corresponding response, change. The hypertext is the engine that controls the available options depending on the application's state.

2.4.3. Constrained Application Protocol (CoAP)

CoAP is a web transfer protocol that is defined in RFC 7252 [8]. It provides the REST paradigm even in constrained environments by being a lightweight protocol. The web transfer protocol HTTP exists since the 1990s [25] and uses as an underlying transport layer the Transmission Control Protocol (TCP) [28]. HTTP/1.1 messages are sent as a stream of characters and cannot be divided into smaller messages. That changed with

HTTP/2 and the usage of frames and streams to reduce overhead and increase the overall performance [29]. Nevertheless, even in more recent HTTP versions, the protocol lacks IoT features for machine-to-machine and asynchronous message exchanges. CoAP is a different protocol, and neither a compressed nor an optimized version of HTTP [30]. The basic header of CoAP has a size of four bytes.

CoAP uses the User Datagram Protocol (UDP) as the underlying layer which is a connectionless protocol. UDP does not retransmit lost packages or guarantee that packages will be delivered in the same order in which they were sent [31]. Therefore, CoAP has implemented a basic reliability mechanism that offers simple stop-and-wait retransmission for confirmable messages and duplicate detection. Due to improvements not only in the physical layer for wireless transmission but also in specialized communication protocols for Low-Power Wide Area Network (LPWAN) [32], wireless and connectionless communication has become more reliable [33]. That offered the possibility to replace TCP by using new protocols which implement only the needed features from TCP in a more efficient way (e.g. QUIC in HTTP/3) while using UDP as transport layer [34].

CoAP makes use of the client/server interaction model such as HTTP does: there are CoAP request and response messages. Requests can be either confirmable or non-confirmable. Supported client request methods are GET, POST, PUT and DELETE similar to HTTP request methods. Response codes are used to give the client feedback to his request. The response code classes are as well similar to HTTP but with a dot after the hundred delimiter. HTTP response code `404 Not Found` is equal to CoAP `4.04 Not Found` [35].

In order to discover information about a host e.g. which resources are available, a GET request can be sent to the path prefix `/.well-known` [36]. Technically speaking, the resource `/.well-known/core` is used in CoAP to discover resources that are hosted. By supporting this mechanism, machine-to-machine communication is again improved, due to the delivery of all hosted resources on the queried CoAP server to the client [37].

2.4.4. Extensible Authentication Protocol (EAP)

EAP is an authentication protocol that supports various authentication methods to authenticate a device or a user [38]. EAP can be used over data link layer protocols such as IEEE 802 or Point-to-Point Protocol (PPP) and does not require the Internet Protocol (IP). EAP brings built-in support for retransmission of lost packages and duplicate elimination but does not guarantee the order of the packages.

The EAP conversation can be split up into three phases. The following example will explain the terms *EAP peer*, *EAP authenticator* and *EAP authentication server* in the context of the EAP phases [39]:

Phase 0: Discovery

Phase 1: Authentication - EAP authentication and AAA Key Transport (optional)

Phase 2: Secure Association Protocol - Unicast Secure Association and Multicast Secure Association (optional)

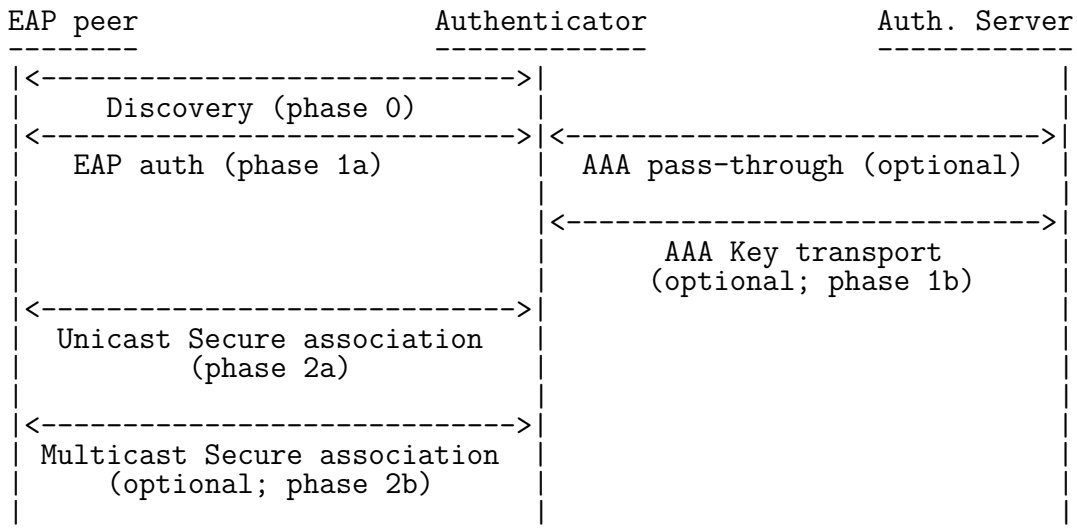


Figure 2.2 - EAP conversation between the parties - Image taken from [39]

EAP is often used for access control in enterprise Wi-Fi networks in combination with a RADIUS server as EAP authentication server [38]. In this context, the EAP peer is the device of the user which asks for network access. The main goal of each EAP

conversation is to establish a secure communication channel between the EAP peer and the EAP authenticator by using fresh key material that is only known to both entities.

When a user tries to connect to a WPA2 Enterprise secured wireless network (Phase 0 completed, the user discovered the emitted Wi-Fi network and the authenticator's capabilities), he is asked to enter his user credentials e.g. username and password when using PEAP as EAP method to authenticate [40].

The term *EAP method* describes, which actual authentication mechanism is used. Popular EAP methods are Protected EAP (PEAP) [41], EAP Transport Layer Security (EAP-TLS) [42], EAP Pre-Shared Key (EAP-PSK) [43] and Nimble out-of-band authentication for EAP (EAP-NOOB) [44].

After discovering the EAP authenticator, the EAP authenticator sends an initial EAP identity request back to the EAP peer who responds with an EAP identity Response message. This message is forwarded by the EAP authenticator to the EAP authentication server which selects the EAP method (in this example PEAP), based on the provided Network Access Identifier (NAI) in the EAP identity request message. The EAP authentication server knows which algorithms and protocols the EAP peer supports by identifying it based on his NAI [45].

The entered credentials by the EAP peer will then be forwarded from the EAP authenticator to the EAP authentication server which will process the request. If the login credentials are correct and the user is allowed to access the network, the peer has been successfully authenticated. Secure Association Protocols e.g. in IEEE-802.11 will only take part between the EAP peer and the EAP authenticator to derive keys and to create a security association (completion of Phases 1 and 2) [46].

2.4.5. Object Security for Constrained RESTful Environments (OSCORE)

OSCORE is a new Internet-Standard, published in the summer of 2019 [22]. Its main purpose is the establishment of a secure communication between two endpoints by providing end-to-end protection using CoAP.

The goal of EAP conversation phase 2 is the creation of a bidirectional security association between the EAP authenticator and the EAP peer. CoAP specifies the use of proxies for efficiency and scalability. The usage of Transport Layer Security (TLS) or Datagram Transport Layer Security (DTLS) in combination with proxies requires its termination at the proxy itself. Despite the advantages in speed and reducing network traffic, (web) proxies have downsides not only in IoT context. They represent a target of choice due to the high impact of a successful attack on that server, because the proxy processes data from many clients in plaintext and messages can be eavesdropped, forged or discarded and impact all connected devices [47].

To mitigate the security concerns when using proxies in the context of IoT, OSCORE protects CoAP requests and responses end-to-end even through different types of CoAP proxies. Among others, the protected message fields are the message payload itself, the request method and the requested resource.

2.4.6. EAP-based Authentication Service for CoAP

In December 2021 version 06 of the IETF-Draft *EAP-based Authentication Service for CoAP*, also known as CoAP-EAP, has been published [48]. It specifies how EAP messages can be exchanged between the IoT device (EAP peer) and IoT controller (EAP authenticator) using CoAP as an EAP lower layer to authenticate an IoT device and grant access to a security domain. The IoT controller can combine the EAP authenticator and the EAP authentication server, but for better understanding, we assume that the IoT controller only has the functionality of the EAP authenticator (see section 2.4.4). Using CoAP as an EAP lower layer does not mean that CoAP is below EAP in the OSI reference model [49]. EAP packages are encapsulated as payload in CoAP messages (application layer) which are then again encapsulated in UDP messages (transport layer).

EAP benefits from the usage of the HATEOAS principle in CoAP, as mentioned in section 2.4.2. CoAP guarantees the order of sent messages which would normally not be possible due to the usage of UDP as transport layer in CoAP. This is done by creating a new resource and making the previous one unavailable in EAP phases 1 and 2 (see figure 2.2).

During EAP phase 1 (authentication) the dynamic resource creation is used to settle on a method. In phase 2 the secure association protocol OSCORE will be established in which the hypermedia indicates the current state of the protocol.

The flow of operation between the IoT device and the IoT controller is specified in the draft:

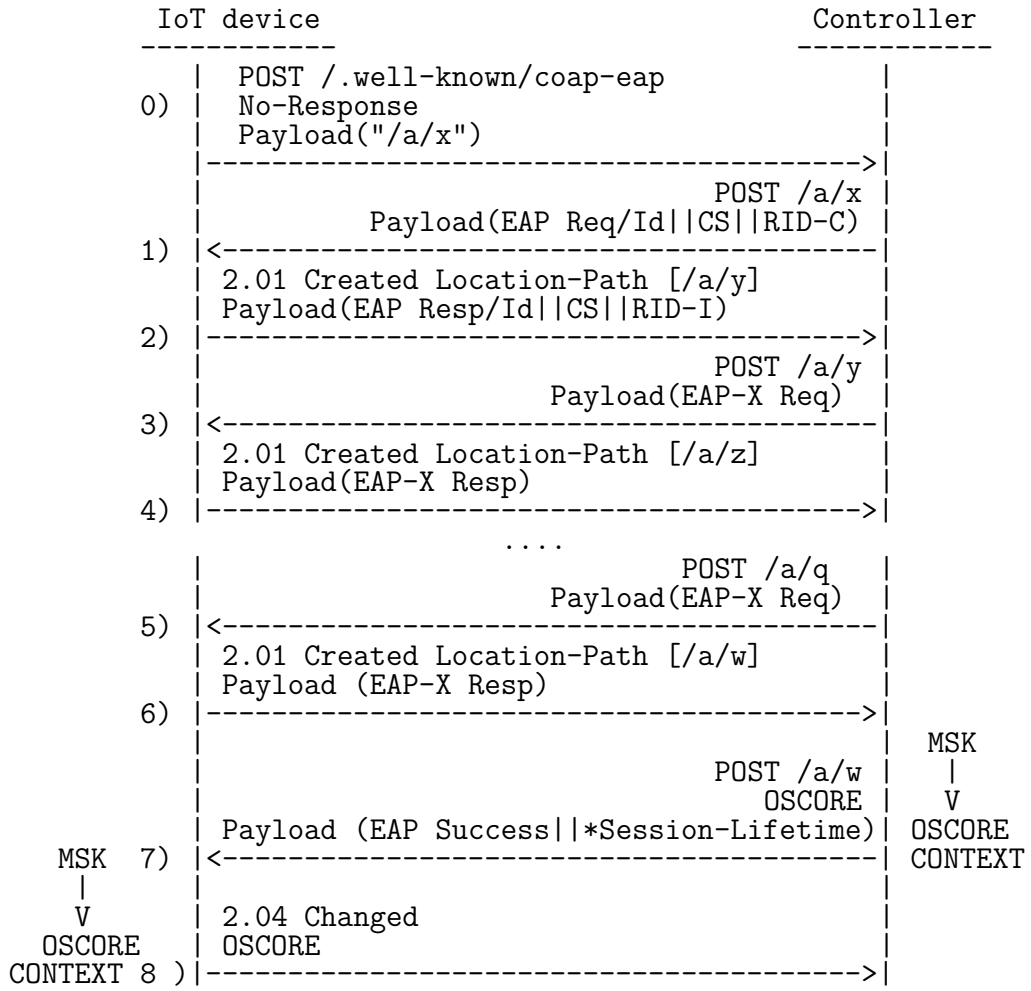


Figure 2.3 - CoAP-EAP flow of operation with OSCORE - Image taken from [48]

The IoT device initializes the authentication process by sending a POST request to the resource `/.well-known/coap-eap` of the IoT controller (step 0). That is the only time, it acts as CoAP client by sending a request. The IoT device tells the IoT controller, which is its initial resource to start the authentication process. In this case, it is the resource `/a/x`. After that initial POST request from the IoT device, it acts as CoAP server by

responding to CoAP requests from the IoT controller (CoAP client) as shown in steps 1 - 7.

The IoT controller then sends a CoAP POST request to `/a/x` (step 1) which contains as payload an EAP request identity message for EAP phase 1. For EAP phase 2, the recipient ID of the Controller (RID-C) and optionally a list with the cipher suites (CS) for OSCORE is sent.

After the IoT device has received this POST request, it passes the received EAP request identity to the EAP peer state machine which returns an EAP response. The resource `/a/x` will be deleted and no longer available. The IoT device will create a new resource `/a/y` and responds to the POST request against `/a/x` with response code `2.01 Created Location-Path /a/y`. The payload of this message is the EAP response identity given by the EAP peer state machine, the recipient ID of the IoT device (RID-I) and `/a/y` is the new resource on which the IoT device can receive a request (step 2).

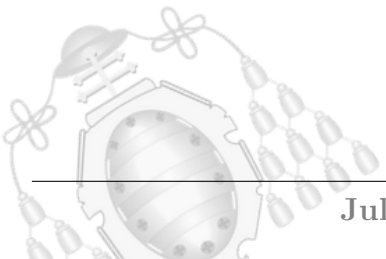
The IoT controller (EAP authenticator) then forwards the EAP response identity to the connected EAP authentication server, where he can choose an EAP method. The EAP authenticator in this case operates as *pass-through authenticator* and supports EAP in pass-through mode. He is capable of forwarding EAP packets from the EAP peer to the EAP authentication server.

EAP peers with different hardware specifications, operating systems and use cases will support a wide variety of EAP methods. In order to indicate that the EAP authentication server chooses one of them, the term *EAP method X* is used. CS and RID-I are not forwarded to the EAP authentication server, because that information belongs to OSCORE and is only used for phase 2, referring to the secure association protocol in figure 2.2.

Steps 3-6 are message exchanges related to the chosen EAP method. As mentioned before, the IoT device (CoAP server) deletes the resource to which a POST request has been received and creates a new one which will be sent in the location header of the CoAP response to the IoT controller (CoAP client). EAP method responses from the EAP peer

are forwarded by the *pass-through authenticator* to the EAP authentication server. In the opposite case, the EAP authenticator forwards EAP method requests from the EAP authentication server destined to the EAP peer as well.

After finishing this process, both the IoT device and IoT controller have negotiated a shared secret, the Master Session Key (MSK). For security reasons, the actual cryptographic operations do not use the MSK itself, but a derivative of it [50]. Here, this derivative is called Master Secret and is used to generate an Object Security Context for Constrained RESTful Environments (OSCORE) (steps 7 - 8). The security services confidentiality, integrity and authentication (see section 2.1) in this constrained environment are therefore achieved even when using proxies through end-to-end protection.



3. Objectives and Methodology of this Work

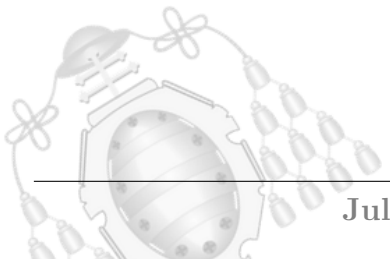
This chapter provides a transition between the explanations in the previous background and the REST implementation in Contiki OS using Erbium. It will describe which parts are in the scope of this document and what methods can be used to achieve these goals.

3.1. Objectives

The main objective of this work is to create web resources dynamically using the Erbium REST engine to implement the HATEOAS principle on the IoT device. The IoT device acts as CoAP server and never sends CoAP requests, except in the first message to initialize the authentication process. Creating dynamic resources is one part needed to create a proof-of-concept implementation compliant with the Internet-Draft *EAP-based Authentication Service for CoAP*.

The concrete goal of this work is to achieve the following objectives:

1. Creating web resources on the IoT device dynamically without assigning memory for the new resources in advance. Sending a CoAP POST request against an existing resource should make the current resource unavailable and create a new random resource during runtime. The response should return the appropriate response code and the URI of the new resource.
2. Sending an initial CoAP POST request from the IoT device to act as CoAP client.
3. Developing a proof-of-concept implementation to demonstrate a CoAP-EAP-based authentication by applying the HATEOAS principle.



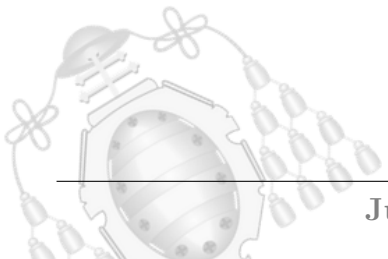
3.2. Methodology

All three objectives require a test setup consisting of an IoT device that is executing the Contiki Operating System and its official library for REST: Erbium. This setup and its deployment will be explained in chapter 4.

To accomplish *Objective 1*, Erbium will be analyzed in detail. This will answer questions about what resources are and how they can be created or deleted. This fundamental understanding is necessary to see if a dynamic creation of resources during runtime is possible and how it could be done. In order to return CoAP status codes and fill other header fields with the appropriate values, Erbium's REST functions will be inspected as well.

Objective 2 changes the focus from responding to CoAP requests to its functionalities as CoAP client to send requests. This contains creating a CoAP message with a payload and sending it to a destination IP address.

To achieve *Objective 3* not only both previous objectives have to be merged. The test setup will be extended with an IoT controller (CoAP-EAP controller) and an EAP authentication server to simulate the authentication of the IoT device. Therefore, an implementation of the EAP state machine (EAP-SM) is used to process incoming EAP messages and generate EAP responses. This will result in the authentication of the IoT device without user interaction after starting the simulation and covers steps 1 - 7 of the CoAP-EAP flow of operation as shown in figure 2.3.



4. Implementation

In this chapter, the underlying mechanism in the Erbium REST engine will be explained. The deployment of motes using the Cooja Simulator is introduced as well as implementation details about how to use the HATEOAS principle in Erbium. The last part of this chapter will explain the ideas and the source code behind the proof-of-concept implementation compliant with CoAP-EAP draft Version 06.

4.1. Contiki OS and the Cooja Simulator

The Contiki Operating System has been specifically designed for the use on low-power microcontrollers. It has been ported to different hardware platforms (such as Freescale MC13224V, Intel 8051-based platforms, ZOLERTIA Z1 and more), supports IPv4 and IPv6 networking protocols and wireless standards for Internet and computer network communication on constrained devices like 6LoWPAN and CoAP [10]. With at least 10 kB RAM and 30 kB ROM hardware requirements it has been designed to operate on very limited devices [51].

Applications for Contiki are written in the programming language C and can make use of dynamic memory allocation, but Contiki does not have a Memory Protection Unit (MPU), because the supported hardware platforms do not offer this functionality [52]. Therefore, the operating system does not separate processes' memory and areas worthy of protection are not safeguarded [53].

In the process of developing applications for IoT devices, the use of network and IoT device simulators offer several advantages, such as low-cost testbed setups on x86 hardware with an almost unlimited amount of IoT devices, quick reboots and advanced debugging interfaces [54]. Cooja is a network simulator for Contiki and developed in Java. The Java Runtime Environment (JRE) allows virtualizing Contiki by using Cooja on different hardware platforms [55]. Simulated microcontrollers inside Cooja are called *motes*. Each

mote compiles and executes the full Contiki Operating System and has access to the supported protocols and standards in Contiki [56], even though Cooja may not be the first choice for testing highly precise time-related performance measurements [57]. Projects in Cooja contain all information about motes with its compiled C code, border routers, IP configuration and distances for wireless networks. They can be exported to a `.csc` file which represents a Cooja scenario. The following image shows a scenario with two motes:

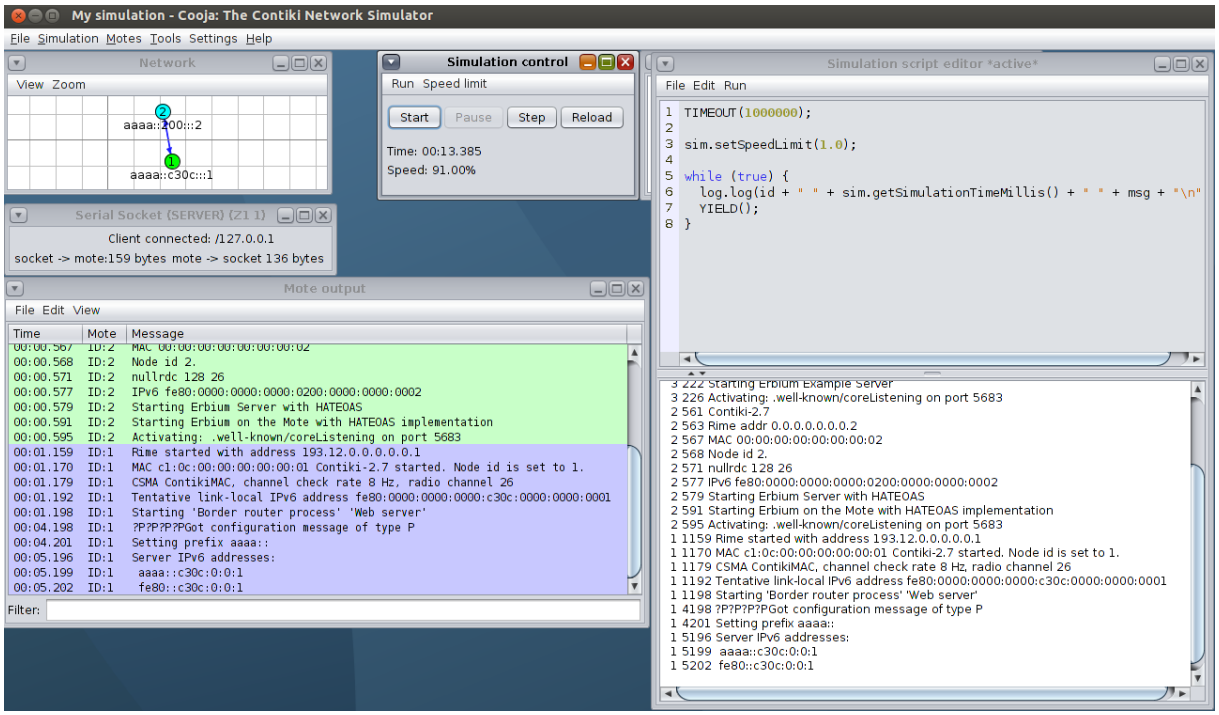


Figure 4.1 - Running scenario in the Cooja Simulator

Mote 1 has the IP address `aaaa::c30c:0:0:1` assigned and represents the border router which is used to route packages between the hypervisor network and the network inside the Cooja simulation [58]. Mote 2 represents an emulated Texas Instruments MSP-EXP430F5438 microcontroller [59] with 256 kB of RAM which loads the Contiki Operating System including all device-specific configurations (see figure 4.2). The device-specific configuration enables Erbium (see section 4.1.1) with self developed web resources as explained in section 4.6, all merged into the file `Erbium-CoAP-EAP-HATEOAS-IoT-Device.c`.

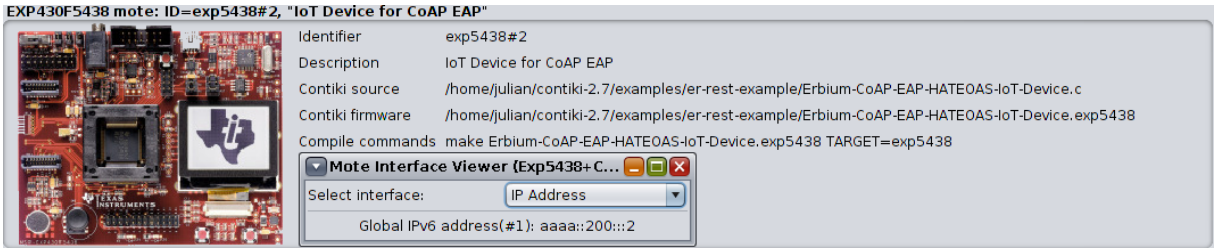


Figure 4.2 - Mote type and loaded .c file

4.1.1. Erbium REST Engine

In order to interact with an IoT device through its offered web resources, a REST engine is needed. One of the provided examples of Erbium includes the following scenario: The IoT device (a mote in the Cooja Simulator) waits for CoAP POST requests against its resource `/actuators/toggle`. If a request is received an event is triggered. In this case, the red LED of the simulated microcontroller will be toggled.

Erbium became the official REST implementation for Contiki in 2011 [9]. The developers provide multiple Erbium example scenarios to interact with resources by using the CoAP [60]. Because of better memory management and efficiency reasons those resources are statically defined. Their URI is hard-coded before starting the mote.

4.1.2. Copper CoAP User-Agent

Sending CoAP requests to a Contiki mote can be realized by using the Copper CoAP User-Agent shown in figure 4.3. It is an extension for Mozilla Firefox and implements CoAP functionalities in JavaScript [61]. Cooja does not offer an integrated web browser. Therefore, the one from the host operating system has to be used. In order to enable communication between the host machine and the mote inside Cooja, a tunslip tunnel is used [62]. Requests to a resource can then be sent by using the following syntax in the address bar of the web browser `coap://[aaaa::200:0:0:2]:5683/resource`, assuming that the destination IP address `aaaa::200:0:0:2` and the default CoAP UDP port 5683 is used.

In order to discover all available resources, a GET request can be sent to the resource `/.well-known/core` as explained in section 2.4.3, which is done by using the button *Discover* in Copper:

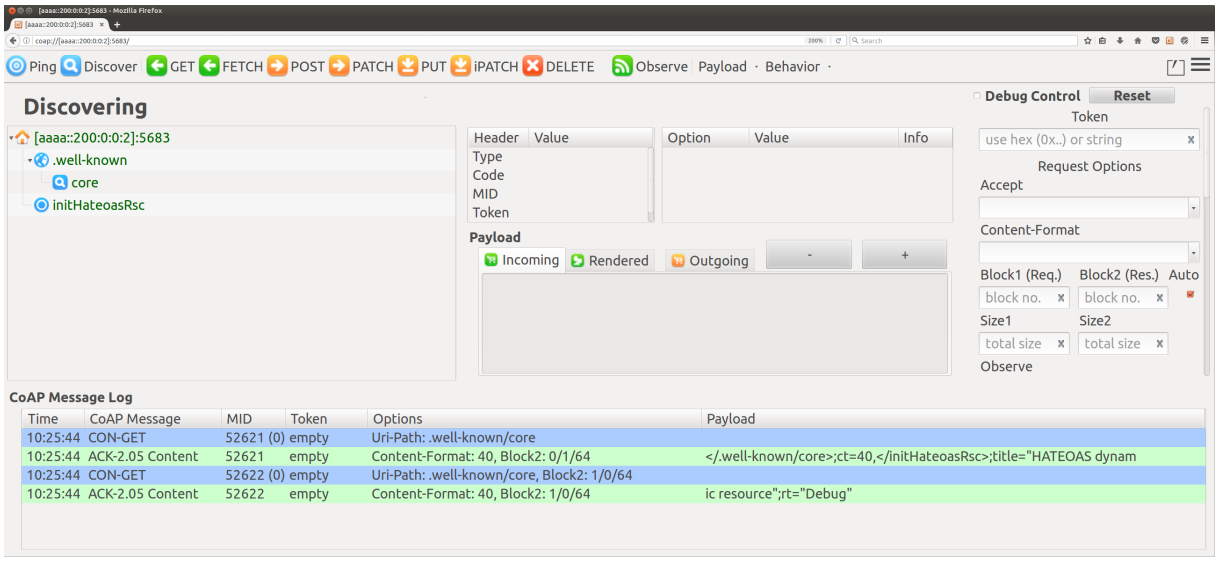


Figure 4.3 - Discovering the available resources on the mote

4.2. Deployment

During the process of work with the Erbium REST engine, Cooja was executed on a Ubuntu 12.04 32-bit virtual machine. This outdated Ubuntu release includes all required dependencies for the use of Cooja and offered native compatibility with the Copper (Cu) CoAP user-agent due to its old Mozilla Firefox version. Since Firefox 57 protocol handler extensions are not supported anymore [63]. The Cooja installation is explained in the corresponding wiki [64]. In this work, Contiki release 2.7 was used. After installing all dependencies, the scenario `server-client.csc` can be run as described by the developers [60].

In order to achieve *Objective 1* of this work, first only the `er-example-server.c` file has been modified to analyze the behavior of the REST engine and to create resources dynamically.

Objective 2 and *Objective 3* did not rely anymore on the `server-client.csc` example scenario. A new Cooja scenario had been created and the mote type had been changed to a

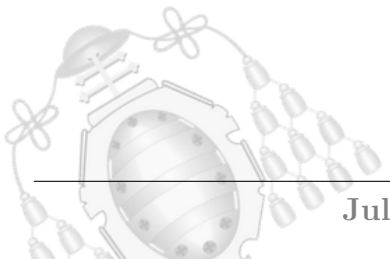
Texas Instruments MSP-EXP430F5438 as shown in figure 4.2, due to memory limitations on the mote in the example scenario. A CoAP-EAP controller was used as an IoT controller as shown in figure 2.3 to perform the authentication of the IoT device. The IoT controller acted as a RADIUS client to interact with the running RADIUS server. The RADIUS server takes the role of the EAP authentication server in EAP terminology (see figure 2.2). Both the CoAP-EAP controller and the RADIUS server were running on the same Ubuntu VM to support EAP in pass-through mode as explained in section 2.4.6.

Due to security concerns, this Ubuntu version should not be used in a productive and Internet-connected environment [65].

4.3. Erbium Example Scenario and the CoAP-EAP Flow of Operation

In the example scenario `server-client.csc` two Motes are generated: The Erbium example server and the example client. Both motes load the Contiki Operating System by using the corresponding C files `er-example-client.c` [66] and `er-example-server.c` [67]. The client mote is used to generate and send CoAP requests to the server mote which will process those CoAP requests and confirms to the client that the message has been received, if it was from type confirmable.

In the given scenario, the client toggles the red LED on the server mote every 10 seconds by sending CoAP POST requests. As seen in the CoAP-EAP flow of operation in figure 2.3, the IoT device acts only in step 0 as CoAP client by sending a POST request to the resource `/.well-known/coap-eap`. In all further steps, the IoT device acts as a CoAP server. It receives CoAP requests from the IoT controller, deletes the old resource, creates a new one and responds with code *2.01 Created*. The path of the new resource is sent as well with the response code in the location-path CoAP option. In order to implement this wanted behavior, it is important to understand how resources are handled in Erbium.



4.4. Understanding Resources in Erbium

Only the CoAP server implementation deals with the generation of resources. Therefore, the implementation is inside `er-example-server.c`. In this section, the predefined resource `/hello` is used to explain what is needed to create a resource.

4.4.1. C Preprocessor Directives

An object-like macro is used to enable a resource that is activated by using the define directive: `#define REST_RES_HELLO 1`.

In this case, the C preprocessor will recognize each occurrence of `REST_RES_HELLO` and replaces it with `1` [68]. Because of that, the code in between `#if REST_RES_HELLO` and `#endif` will be executed [69].

```
#if REST_RES_HELLO
// code in here will be executed
// only when REST_RES_HELLO had been replaced by 1.
// it is used to expand the resource macro,
// defines the resource handler function
// and set REST parameters.
#endif
```

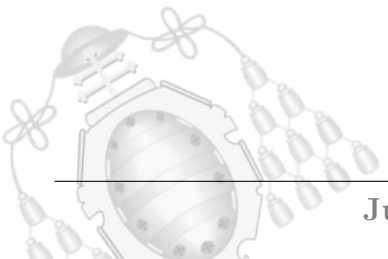
Figure 4.4 - Usage of object-like macro in C

4.4.2. Erbium's Resource Macro

Creating a resource in Erbium requires the expansion of the *RESOURCE* macro in Erbium. The definition can be found in the file `erbium.h` [70]:

```
#define RESOURCE(name, flags, url, attributes) \
void name##_handler(void *, void *, uint8_t *, uint16_t, int32_t *); \
resource_t resource_##name = {NULL, flags, url, attributes, name##_handler, NULL, NULL,
    NULL}
```

Figure 4.5 - Definition of a resource in the file erbium.h



In order to create the resource `/hello`, the resource macro will be expanded with the arguments shown in figure 4.6. It is placed inside define directive `REST_RES_HELLO`. The leading forward slash in the third parameter *hello* can be omitted:

```
RESOURCE(helloworld, METHOD_GET, "hello", "title=\"Hello world: ?len=0..\";rt=\"Text\"");
```

Figure 4.6 - Expanding the resource macro to create the helloworld resource

Expanding the RESOURCE macro first declares a function `helloworld_handler` from return type `void`. Its actual implementation has to be placed inside define directive `REST_RES_HELLO` as well. After that, a structure variable `resource_helloworld` from the datatype `resource_t` is initialized with its passed parameters by using the token pasting operator [71]. The C preprocessor will generate the following code [72]:

```
    void helloworld_handler(void *, void *, uint8_t *, uint16_t, int32_t *); \
    resource_t resource_helloworld = {NULL, METHOD_GET, "hello", "title=\"Hello world: ?len
    =0..\";rt=\"Text\"", helloworld_handler, NULL, NULL, NULL}
```

Figure 4.7 - The expanded resource macro for the helloworld resource

4.4.3. Data Structure for a Resource in REST

The corresponding declaration for a data structure from datatype `resource_t` can be found as well in `erbium.h` [70]:

```
struct resource_s {
  struct resource_s *next; /* for LIST, points to next resource defined */
   rest_resource_flags_t   flags; /* handled RESTful methods */
  const char* url; /*handled URL*/
  const char* attributes; /* link-format attributes */
   restful_handler handler; /* handler function */
   restful_pre_handler pre_handler; /* to be called before handler, may perform initializations */
   restful_post_handler post_handler; /* to be called after handler, may perform finalizations (
     cleanup, etc) */
  void* user_data; /* pointer to user specific data */
  unsigned int benchmark; /* to benchmark resource handler, used for separate response */
};
typedef struct resource_s resource_t;
```

Figure 4.8 - Declaration of the struct resource_t

Expanding the resource macro also causes the initialization of the declared structure variable `resource_helloworld` with the passed arguments. A structure variable with the name `helloworld` from the datatype `resource_t` will then have multiple elements. The concrete code will look as follows:

```
resource_t helloworld = {
  NULL,
  METHOD_GET,
  "hello",
  "title =\"Hello world: ?len=0..\";rt=\"Text\"",
  helloworld_handler,
  NULL,
  NULL,
  NULL
};
```

Figure 4.9 - Initializing the struct variable helloworld for the helloworld resource

The third element in this struct with the name `url` is a pointer to a constant char: `const char* url`. After the initialization, this pointer will reference to a char-array which will contain in each array element the letters of the string "hello", followed by the null character `\0` to mark the end of a multibyte character string [73].

4.4.4. The Resource Handler Function

For each resource name (the first parameter when expanding the resource macro from `erbium.h`) a handler function with the name `[resourcename]_handler` must be implemented inside the define directive `REST_RES_HELLO`. The code inside the handler function will be executed when the resource is accessed by the defined CoAP request method. The second element in the struct from type `resource_t` determines the CoAP request method.

The functionality inside the handler function can be for example only one line with a function call to toggle the red led on the server mote `leds_toggle(LEDS_RED);` or defining the content types and response payloads for the used method. For returning the newly created location path especially those REST functions become important.

4.4.5. Activating the Resource

Even though all previous steps have been done (expanding the macro by using the define directive and implementing a handler function), the resource will not be accessible. Only resources which had been activated will be accessible.

In order to enable the resource `/hello` from the helloworld example, a call by reference to the memory address of the structure variable `resource_helloworld` is used, which had been initialized during macro expansion. This activation can be linked to the define directive `REST_RES_HELLO` (see figure 4.5) to active the resource after initializing the REST engine.

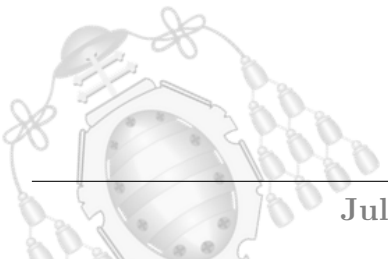
```
1  ...
2  rest_init_engine ();
3  #if REST_RES_HELLO
4     rest_activate_resource (&resource_helloworld);
5  #endif
6  ...
```

Figure 4.10 - Activating the resource helloworld

4.5. Creating Dynamic Resources

Resources in Erbium are statically defined and Erbium does not offer a pre-built method to create a new resource and delete the previous one during runtime. They are defined before starting the mote. In order to implement the HATEOAS principle as required for the latest version 06 of the *EAP-based Authentication Service for CoAP* (see figure 2.3), Erbium's resource and resource handler function have to be adapted.

Figure 4.11 shows the flow of packages when sending POST requests to the IoT device in a schematic way. The mote is running the Contiki Operating System with an adapted version of the `er-example-server.c` file. The source code is explained in section 4.5.1. The POST requests are sent by using the Copper Plugin to see if the behavior is as expected to achieve *Objective 1* from section 3.



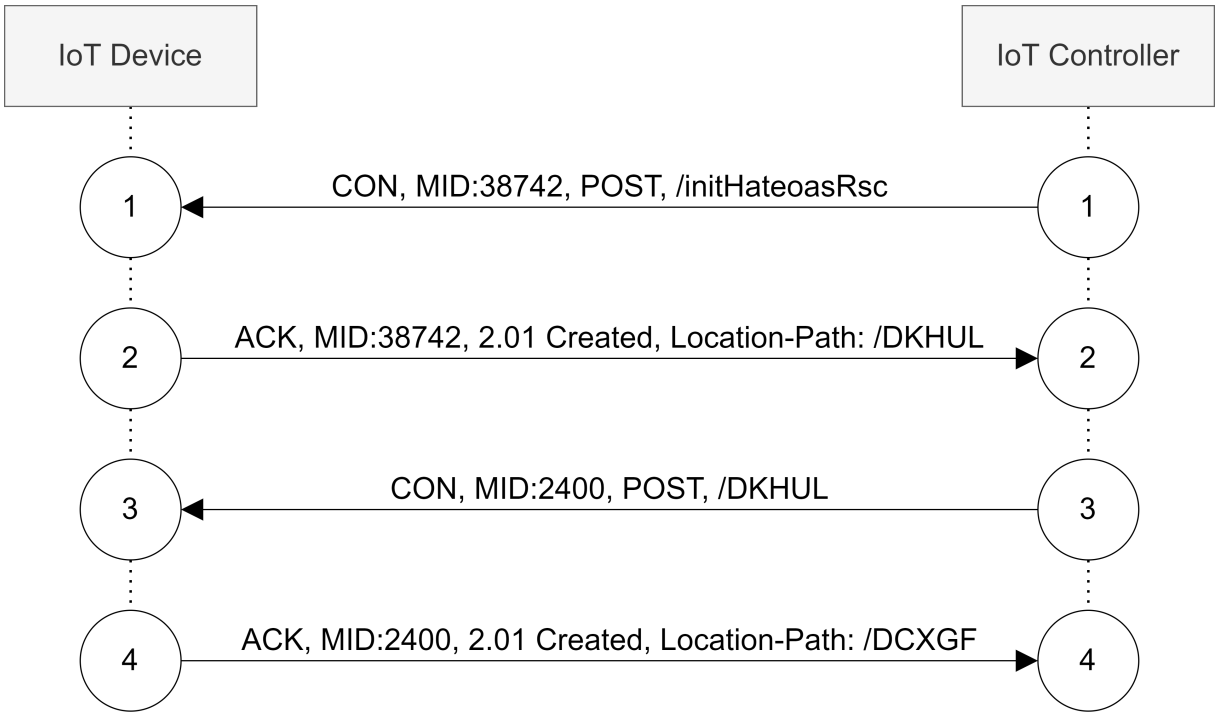


Figure 4.11 - Schematic flow of packages in dynamic resource creation

The first message from the IoT controller to the IoT device is a CoAP POST request against the initial and static resource `/initHateoasRsc`.

The initial request will be responded by the IoT device with response status code *2.01 Created* and the location path `/DKHUL` (five random uppercase characters) of the newly created and available resource. A POST request to `/initHateoasRsc` will now return *4.04 Not Found*.

The second POST request from the IoT controller to the IoT device is now sent against the new resource `/DKHUL` of the IoT device. Again, the IoT device responds with status code *2.01 Created* and a new location path `/DCXGF`. Neither `/initHateoasRsc` nor `/DKHUL` are longer available.

4.5.1. Implementation Details on Updating an Initial Resource

The source code in figure 4.12 has been developed as part of this thesis and uses the Erbium REST engine to achieve *Objective 1*. The implementation shows the wanted

behavior as seen in the schematic flow of packages (see figure 4.11) in the process of dynamic resource creation when sending POST requests against a resource. In order to enable this resource, the object-like macro has to be expanded by using a new define directive for this resource: `#define REST_RES_HATEOAS 1`. Then the resource macro will be expanded and the REST resource will be activated, analog to the example resource helloworld in section 4.10.

The resource name `hateoas` is used in section 4.6 as well to merge the dynamic resource creation with sending an initial POST request and the communication with the EAP State machine. This achieves *Objective 3* by providing a proof-of-concept that showcases steps 0 - 7 on the IoT devices side in the CoAP-EAP flow of operations (see figure 2.3).

```
1  #if REST_RES_HATEOAS
2  char urlString [] = "initHateoasRsc";
3  int urlStringCounter;
4
5  RESOURCE(hateoas, METHOD_POST, urlString, "title=\"HATEOAS dynamic resource\";rt=\"
       Debug\"");
6
7  void hateoas_handler(void *request, void *response, uint8_t *buffer, uint16_t preferred_size,
       int32_t *offset) {
8    for(urlStringCounter = 0; urlStringCounter < 5; urlStringCounter++) {
9      urlString[urlStringCounter] = 'A' + (random_rand() % 26);
10   }
11   urlString[5] = '\0';
12
13   REST.set_response_status(response, REST.status.CREATED);
14   REST.set_header_content_type(response, REST.type.TEXT_PLAIN); /* text/plain is the default,
       hence this option could be omitted. */
15   REST.set_header_location(response, resource_hateoas.url);
16 }
17 #endif
```

Figure 4.12 - Implementation for dynamic resources

First, an initial static resource is created by expanding the resource macro (line 5). Unlike the default resources, the macro arguments are used differently. The `url` argument is not directly used by entering a string (compare with figure 4.6, but by using a char-array called `urlString` (line 2) which will later be modified. Its initialization was done by specifying an initial string literal enclosed in double quotation marks to ensure, that the null character `\0` is included at the end of the string to indicate its termination.

Assigning a new string literal to this already initialized char-array with the assignment operator will not work, due to restrictions in C [74]. Therefore, a for loop was used to change the first five characters by altering each array element individually from `urlString[0]` to `urlString[4]` (line 9) with a new random uppercase character and terminating the string after that (line 11).

A loop variable `urlStringCounter` was defined outside of the for loop due to the restriction that initial loop declarations inside the for loop initialization are only allowed in C99 mode [74]. The generation of the new random uppercase character inside this for loop makes use of character arithmetic in C. The implementation of `random_rand()` in Contiki returns a pseudo-random number between 0 and 65535 [75]. The character `A` is stored by its decimal representation 65 according to the ASCII implementation on Linux [76]. To generate a new random uppercase character for the elements `urlString[0]`, `urlString[1]`, ... `urlString[4]` the following calculation is done:

$$65 + (x \bmod 26) = y \text{ with } \{\, x \mid x \in \mathbb{N}, x \in [0, 65535] \,\} \text{ and } \{\, y \mid y \in \mathbb{N}, y \in [65, 90] \,\}$$

Figure 4.13 - Calculations for character arithmetic in C

Therefore the saved decimal character between 65 and 90 will be interpreted as uppercase characters from A to Z.

The size of the char-array cannot be modified after its initialization even though it contains less characters. For `urlString[]` will be always (14+1)∗sizeof(char) = 15 byte of memory reserved, assuming that sizeof(char) = 1 byte. The string `initHateoasRsc` has a length of 14 characters and the additional character is caused by the null terminator character at the end of the string.

After all character operations, the resource has a new name because of the changes in the underlying data structure representing a REST resource as seen in 4.8. The struct element `const char* url` inside the variable `resource_hateoas` from type `resource_t` has not changed its position in memory, but only some characters of the char-array have changed.

It is required that the IoT controller will be informed about the newly created location path. This behavior is compliant with the CoAP specification as explained in section 2.4.3. Thus, the response in the correct format needs to be generated. This is done by responding with response status code *2.01 Created* for a created resource (line 13), defining the rest implementation type to `text / plain` (line 14) and setting the location option of the response to the updated char-array `urlString[]` (line 15).

4.5.2. Keeping the old resources

During the process of developing the implementation of new dynamic resources, an attempt was made to keep the old resources. For the creation of n resources and keeping all those n resources, either memory for n resources needs to be assigned before executing the program or dynamically when the request of creation is done. Due to the low amount of memory (256 kB on the TI MSP430F5438), it is not reasonable to assign memory in advance for even a few resources that might be created.

Therefore, the usage of dynamic memory allocation would be needed. Contiki supports multiple ways to allocate and deallocate memory on the heap: the standard C library memory allocator function `malloc` [77], the `mmem` managed memory allocator and the `memb` memory block allocator [78]. The usage of `malloc` is not recommended on IoT devices due to memory fragmentation.

The most often used memory block allocator is `memb` which stores objects of constant size in static memory.

Having said that, keeping (an unlimited amount of) old resources in memory will consume more and more memory over the time of the EAP exchange. This reduces the uptime of the device due to the high costs of dynamic storage allocation and could cause device crashes when running out of memory [79]. Keeping the old resources is neither compliant with the CoAP-EAP draft, nor in this case useful.

4.6. Proof-of-Concept Implementation for Draft Version 06 of CoAP-EAP

In section 4.5.1, the dynamic creation of resources had been implemented by using the Erbium REST engine. To provide a proof-of-concept compliant with CoAP-EAP draft version 06 (see figure 2.3), the IoT device has to send an initial POST request against the IoT controllers resource `/.well-known/coap-eap` in step 0. The payload of this message contains the IoT devices resource, under which it will respond to an incoming CoAP POST request. In this implementation, it will be the initial and static resource `/initHateoasRsc`. From that point on the IoT device will create a new resource (and delete the old one as explained in section 2.4.6) after receiving a POST request against this resource. This behavior will be maintained until the IoT device receives an EAP Success message in the CoAP payload. Then the MSK is available and OSCORE will be established.

The full source code which is executed on the IoT device is attached in appendix A.2 of this document. The explanation in the following sections will refer to the lines in the appended source code.

4.6.1. Sending the Initial POST Request

After initializing the REST engine (line 292), a timer is set to wait 15 seconds (line 303) until the application-specific events are executed. For this implementation, only one event is needed which is sending a CoAP request to the IoT controller's IPv6 address (line 85). To ensure, that this request is only sent once, an if statement is used not only to check if the timer is expired but also if the initial request was only sent once (line 313).

A CoAP message is created in different steps. First, a general struct element of type `coap_packet_t` is created: `static coap_packet_t initialRequest[1];` (line 315). It builds the base of a CoAP packet. The declaration can be found in the Contiki repository under `apps\rest-coap\coap-common.h`. The function `coap_init_message` will take this packet as first argument, sets the `coap_message_type_t` to non-confirmable and sets the the message type to a CoAP POST request (line 319). The destination URI of this request

will be the IoT controllers resource `/.well-known/coap-eap` (line 321). The payload of this message contains the resource under which the IoT device will wait for a request (line 142) and is passed as an argument to the function `coap_set_payload` (line 323). The size of this resource is calculated here to allow a change in the length of the initial resource without the need of changing multiple parts in the source code.

The message will be sent to the destination's pre-defined IPv6 address and UDP port (lines 85 and 86) by using them, just as the CoAP packet itself, as arguments to expand the `COAP_BLOCKING_REQUEST` macro (line 325). This achieves *Objective 2* of this work. Erbium does not offer a built-in macro for sending CoAP requests without expecting a response. Therefore, a blocking request will be sent.

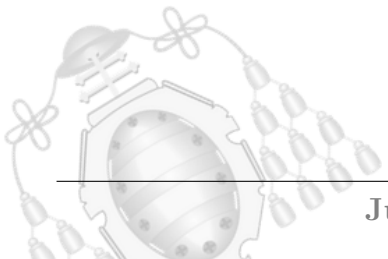
After sending the initial CoAP POST request, the IoT device acts only as a CoAP server. It responds to received requests. When a POST request against the currently alive web resource is received, the `hateoas_handler` function is called (line 150). The resources name is defined by the char-array `urlString` (line 142).

4.6.2. Communicating with the EAP State Machine

To authenticate the device by using an EAP method, the EAP state machine (EAP-SM) has to be initialized (line 278). The used implementation of the EAP-SM [80] in Contiki came from the EAP-related project called *PANATIKI* [81]. Details of the EAP-SM are not part of this work. Only needed details are covered. The communication with the EAP-SM takes place by passing the CoAP payload of an incoming POST request as an argument to the function `eap_peer_sm_step` (line 187). The processed output of the EAP-SM is accessible through the variable `eapRespData`. When the EAP authentication ends successfully, the IoT device receives `EAP Success` in the payload of a CoAP POST request. This will set `eapKeyAvailable` to true. Due to an error in the used EAP-SM library, the IoT controller retransmits the first CoAP POST request, because the corresponding response does not contain the motes NAI as payload. Therefore, the initial resource has to be maintained for the first two received POST requests. This is done by

using the variable `skipFirstHandlerCall` (line 146) as well as condition to change the resources name (line 163). The used EAP method in this proof-of-concept is EAP-PSK.

In step 1 of the CoAP-EAP flow of operation shown in figure (2.3) the Controller sends an EAP request identity message (see section 2.4.6). As a response to this message, the IoT device will once send its supported ciphersuites to the controller. This functionality is managed by the variable `cipherSuitesSent`. It is initialized with the value 0, because the ciphersuites are not sent yet (line 66). Therefore, the ciphersuites are sent in the else block (starting in line 210) only one time, because after sending them `cipherSuitesSent` is set to 1 (line 229) and the if block starting in line 191 is executed. The mentioned if block starting in line 191 is executed for each incoming POST request, until the IoT device received the EAP Success message. In this case the EAP-SM returned `eapKeyAvailable = 1` and the OSCORE dummy payload is set (line 210).



5. Results

This chapter presents the results of a proof-of-concept CoAP-EAP implementation with the usage of the Erbium REST engine. Two Wireshark captures will be introduced. The first scenario presents a lossless communication between the IoT device and the IoT controller. The second scenario shows how Erbium handles retransmission due to packet loss. After that, the RAM and ROM usage will be analyzed and the results will be discussed.

5.1. Overview

The testbed for this proof-of-concept implementation consists of multiple parts. The compiled binary from the source file `Erbium-CoAP-EAP-HATEOAS-IoT-Device.c` (see appendix A.2) has the filename `Erbium-CoAP-EAP-HATEOAS-IoT-Device.exp5438`. It contains the Contiki Operating System and the Erbium REST Engine. For the latter, a customized resource and resource handler function is used to support dynamic resource creation needed for the HATEOAS principle (see 4.5.1) and interaction with the EAP state machine. The file is executed on a Texas Instruments MSP-EXP430F5438 microcontroller (see section 4.2).

In order to perform the EAP-based authentication service for CoAP, a Linux implementation of a CoAP-EAP controller was used to take part of the EAP authenticator (see section 2.4.6). This device is the so-called IoT controller as shown in figure 2.3 and has the IP address `aaaa::ff:fe00:1` in this scenario. A Linux RADIUS server acted as EAP authentication server in EAP pass-through mode (see figure 2.2 and section 2.4.6) [82].

The Cooja simulation contains a border router (see 4.1) with the IP address `aaaa::200:0:0:1` and the IoT device with the IP address `aaaa::200:0:0:2`. The border router is used to route packages between the IoT device inside the Cooja network and the

CoAP-EAP controller outside of the Cooja network. In order to establish the network communication, a tunslip tunnel was used [62]. The traffic has been captured with Wireshark to show the message exchanges.

5.2. Wireshark Analysis without Package Loss

The first message from the IoT device to the IoT controller is a CoAP POST request against the resource `/.well-known/coap-eap` (frame number 538) to start the authentication process. This initial request will not be responded but causes the IoT controller to send his first CoAP POST request to the IoT device against its initial and static resource `/initHateoasRsc` (frame number 539).

No.	Source	Destination	Protocol	Length	Info
538	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	105	NON, MID:62710, POST, /.well-known/coap-eap
539	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	98	CON, MID:1, POST, TKN:3e c6 6e 35, /initHateoasRsc
558	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	111	ACK, MID:1, 2.01 Created (text/plain), TKN:3e c6 6e 35, /initHateoasRsc
578	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	98	CON, MID:1, POST, TKN:3e c6 6e 35, /initHateoasRsc [Retransmission]
751	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	99	ACK, MID:1, 2.01 Created (text/plain), TKN:3e c6 6e 35, /initHateoasRsc
756	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	108	CON, MID:2, POST, TKN:3e c6 6e 35, /DKHUL
770	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	140	ACK, MID:2, 2.01 Created (text/plain), TKN:3e c6 6e 35, /initHateoasRsc
773	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	138	CON, MID:3, POST, TKN:3e c6 6e 35, /CXGFE
780	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	123	ACK, MID:3, 2.01 Created (text/plain), TKN:3e c6 6e 35, /initHateoasRsc
783	aaaa::ff:fe00:1	aaaa::200:0:0:2	OSCORE	104	CON, MID:4, POST, TKN:3e c6 6e 35, /SXCLC
797	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	72	ACK, MID:4, 2.05 Content, TKN:3e c6 6e 35, /initHateoasRsc

Figure 5.1 - Wireshark capture filtered to CoAP messages without package loss in the proof-of-concept implementation

The *Info* column contains a summary of a captured package. In Wireshark version 3.6.5, it shows the URI path at the end of each captured CoAP message. Wireshark indicates the same URI path `/initHateoasRsc` in all responses to previous CoAP POST requests from the IoT controller (frames 751, 770, 780, 797), because the token `3ec66e35` did not change. The token is a field in the header of a CoAP message [8] and is used as an identifier during the whole authentication process. Therefore, Wireshark displays the initial resource instead of the location path at the end of the mentioned messages. The URI path is not sent in those messages but shown in Wireshark as an interpretation of this communication.

The IoT device acknowledges the request (frame number 539) by sending a response with response status code *2.01 Created* and the new location path `/DKHUL` (five random uppercase characters) of the newly created and available resource (frame number 751).

Figure 5.2 shows this frame in detail. A POST request to `/initHateoasRsc` would now return status code *4.04 Not Found*.

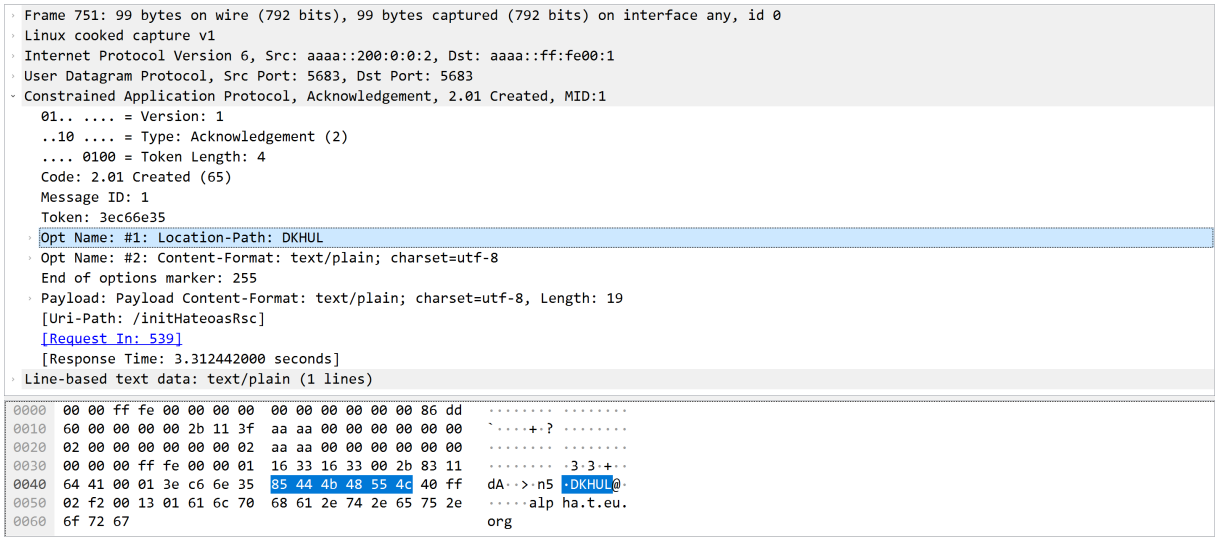


Figure 5.2 - Wireshark analysis of frame 751

Due to the mentioned error (see section 4.6.2) in the EAP-SM on the IoT device, the response to the request against `/initHateoasRsc` contains an incorrect payload. Therefore, frame 578 is a retransmission of frame 539 but handled by creating dynamic resources only from the second incoming message (see section 4.6.2). The controller chose *EAP-PSK* as EAP method and from now on EAP packets related to this method are exchanged between the IoT device and the IoT controller in the CoAP message payload.

Frame number 756 is the first EAP-PSK-related request against `/DKHUL`. The IoT device responds in frame number 770 with the processed information by the EAP-SM in its payload. The location path changed to `/CXGFE`. Then, another exchange regarding this EAP method is done (frames 773 and 780) and the location path is finally changed to `/SXCLC`.

After finishing the EAP authentication with success, the controller changes the CoAP option to *OSCORE*. The payload of this request against `/SXCLC` contains *EAP Success*. Figure 5.3 shows details about this frame. This behavior is as required in step 7 in figure 2.3.

Figure 5.4 shows the console output of the CoAP-EAP controller.

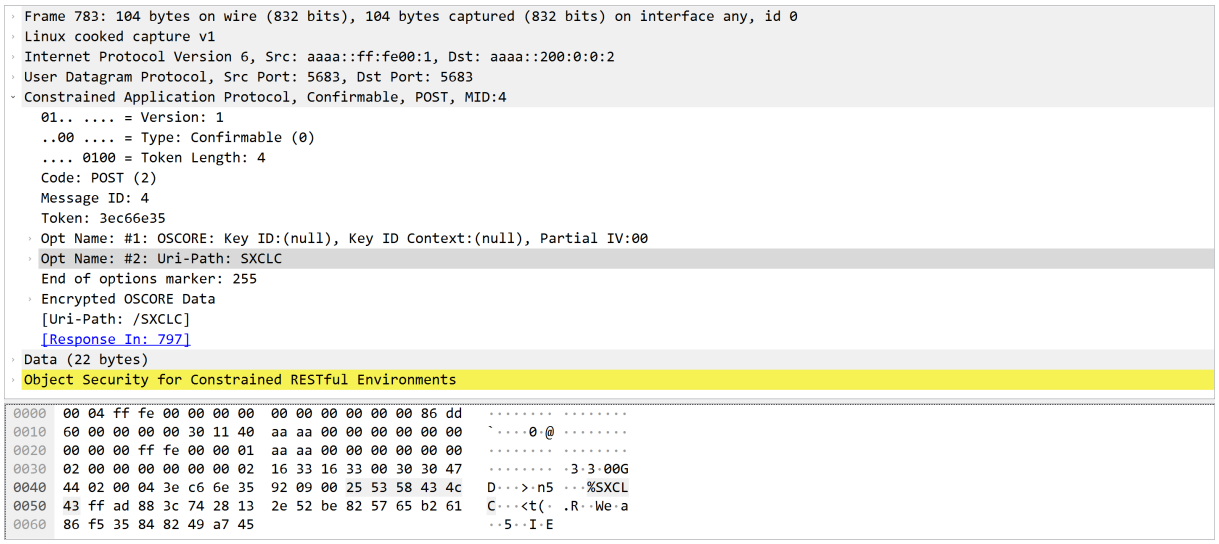


Figure 5.3 - Wireshark analysis of frame 783

```
decapsulated EAP packet (code=3 id=90 len=4) from RADIUS server: EAP Success
EAP: EAP entering state SUCCESS2
EAP SUCCESS::::::::::::::::::::::
```

Figure 5.4 - Console output of the CoAP-EAP controller

From this point on, the communication is secured with OSCORE. OSCORE already protects frame number 797 and all ongoing packages between the IoT device and the IoT controller (see chapter 2.4.5). The last used resource `/SXCLC` will be maintained and not changed.

5.3. Wireshark Analysis with Package Loss

The dynamic creation of resources on the IoT device results in status code *4.04 Not Found* if two different requests are sent against the same resource. The first request against `/CJSBK` in figure 5.5 returns the location path of the new resource `/VWPUD`. All future requests against `/CJSBK` will result in status code *4.04 Not Found* (figure 5.6). The POST requests have been sent with Copper as explained in section 4.1.2.

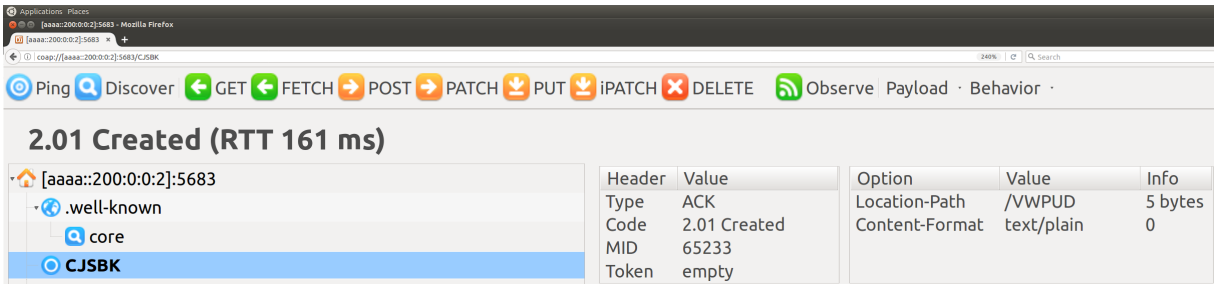


Figure 5.5 - Sending a first POST request against /CJSBK returns 2.01 Created

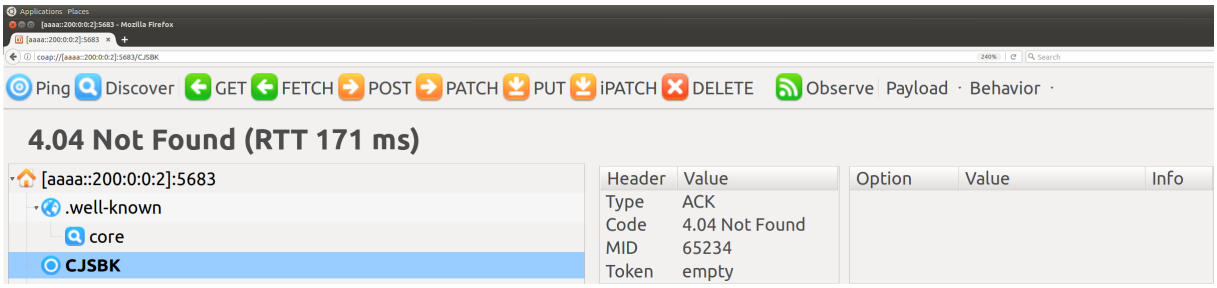


Figure 5.6 - Sending another POST request against /CJSBK returns 4.04 Not Found

Package loss can be simulated with Cooja as well. The Cooja scenario for the second Wireshark capture was configured with a success probability of 50% for inbound and outbound data for motes that use the Cooja radio medium. Figure 5.7 shows the capture of the same communication between the IoT device and the IoT controller as in section 5.2, but with the configured package loss. Therefore, the location paths are different compared with figure 5.1 due to their random creation.

No.	Source	Destination	Protocol	Length	Info
956	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	105	NON, MID:62710, POST, /.well-known/coap-eap
973	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	111	ACK, MID:1, 2.01 Created (text/plain), TKN:0c 47 08 78, /initHateoasRsc
986	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	98	CON, MID:1, POST, TKN:0c 47 08 78, /initHateoasRsc
1164	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	99	ACK, MID:1, 2.01 Created (text/plain), TKN:0c 47 08 78, /initHateoasRsc
1169	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	108	CON, MID:2, POST, TKN:0c 47 08 78, /GDKHU
1176	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	140	ACK, MID:2, 2.01 Created (text/plain), TKN:0c 47 08 78, /initHateoasRsc
1179	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	138	CON, MID:3, POST, TKN:0c 47 08 78, /EDCXG
1189	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	138	CON, MID:3, POST, TKN:0c 47 08 78, /EDCXG [Retransmission]
1208	aaaa::ff:fe00:1	aaaa::200:0:0:2	CoAP	138	CON, MID:3, POST, TKN:0c 47 08 78, /EDCXG [Retransmission]
1215	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	123	ACK, MID:3, 2.01 Created (text/plain), TKN:0c 47 08 78, /initHateoasRsc
1218	aaaa::ff:fe00:1	aaaa::200:0:0:2	OSCO…	104	CON, MID:4, POST, TKN:0c 47 08 78, /PSXCL
1225	aaaa::200:0:0:2	aaaa::ff:fe00:1	CoAP	72	ACK, MID:4, 2.05 Content, TKN:0c 47 08 78, /initHateoasRsc

Figure 5.7 - Wireshark capture filtered to CoAP messages with package loss in the proof-of-concept implementation

Figure 5.8 shows frame number 1208 in detail. It is a POST request from the IoT controller to the IoT device. It was resent two times because the corresponding response to neither

frame 1179, nor frame 1189 could not be delivered to the IoT controller. Those messages got lost due to transmission errors but do not interfere with the authentication process.

```
Frame 1208: 138 bytes on wire (1104 bits), 138 bytes captured (1104 bits) on interface any, id 0
Linux cooked capture v1
Internet Protocol Version 6, Src: aaaa::ff:fe00:1, Dst: aaaa::200:0:0:2
User Datagram Protocol, Src Port: 5683, Dst Port: 5683
Constrained Application Protocol, Confirmable, POST, MID:3
  01.. .... = Version: 1
  ..00 .... = Type: Confirmable (0)
  .... 0100 = Token Length: 4
  Code: POST (2)
  Message ID: 3
  Token: 0c470878
  Opt Name: #1: Uri-Path: EDCXG
  End of options marker: 255
  Payload: Payload Content-Format: application/octet-stream (no Content-Format), Length: 5
  [Uri-Path: /EDCXG]
  [Response In: 1215]
  [Retransmission of request in: 1179]
Data (59 bytes)
```

Figure 5.8 - Wireshark analysis of frame 1208

Erbium internally recognizes that the request from the Controller was resent due to its handling of CoAP transaction IDs. Therefore, the request will be processed correctly and does not return a resource not found error.

5.4. RAM and ROM Usage

In order to measure the RAM and ROM usage on a Texas Instruments MSP-EXP430F5438 microcontroller, the binary file can be inspected with the command `msp430size Erbium-CoAP-EAP-HATEOAS-IoT-Device.exp5438` [83]. The following table shows a comparison between three executable files containing the full Contiki Operating System:

#	Module	ROM (bytes)	RAM (bytes)
1	Contiki Hello World	19.059	5.072
2	Erbium Hello World	46.067	5.706
3	Erbium CoAP-EAP	51.197	6.498

Figure 5.9 - RAM and ROM usage of an EXP5438 mote running Contiki with different loaded modules

Contiki Hello World represents the compiled binary of the Contiki hello-world example (`hello-world.c`) without networking capabilities [84]. The *Erbium Hello World* executable is a minimized version of the `er-example-server.c` file [60]. All pre-existing resources and their corresponding lines in the source code had been removed except the resource `/hello` as explained in section 4.4.2. Finally, *Erbium CoAP-EAP* in this table represents the binary of the compiled source code shown in appendix A.2. It is the CoAP-EAP implementation compliant with the latest draft version 06. Both Erbium binaries include networking capabilities and the Erbium REST engine itself. On top of that, the CoAP-EAP binary includes the libraries `eap-sm` and `eap-psk` from an EAP-related project called PANATIKI [81].

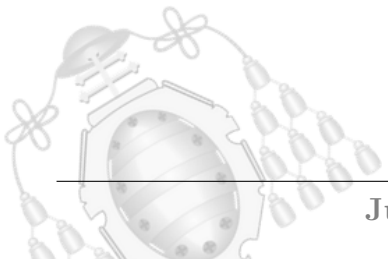
The available memory of 256 kB on the used microcontroller (see figure 4.1) is sufficient for loading the Contiki Operating System and the libraries needed for the CoAP-EAP implementation. Therefore, the mote does not only secure the communication, but has approximately 249 kB of memory left for further application-specific functionalities.

5.5. Discussion

The Cooja mote for executing the proof-of-concept used version 2.7 of the Contiki Operating System. This release received the last changes in the codebase in 2013 [67]. Contiki-NG started in 2017 as a fork of Contiki OS and is still being further maintained today [85]. To run the proof-of-concept with Contiki-NG, the codebase in appendix A.2 has to be adapted due to naming scheme differences. The underlying data structure of resources and how Erbium handles incoming requests did not change. An updated version of the mentioned Erbium example server (see section 4.3) is available in Contiki-NG as well [86] and can be taken as a reference for an adaption.

Among others, the destinations server address and the CoAP message itself were passed as arguments to expand the macro `COAP_BLOCKING_REQUEST` to send a CoAP request with Erbium. Step 0 in the CoAP-EAP flow of operation (see figure 2.2) specifies the usage of the CoAP option *No-Response* for this first message. RFC 7967 introduced the *No-Response* option for CoAP messages to send requests without waiting for a response

[87]. However, the deployed version of Erbium lacks the possibility to set this option. In order to bypass this limitation, the first message was sent from type *Non-confirmable* without the option field set.



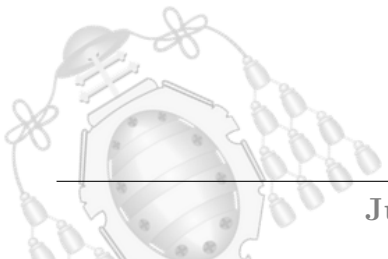
6. Outlook and Conclusions

All three objectives mentioned in chapter 3 have been successfully achieved. The source code for achieving *Objective 3* is attached in appendix A.2. Web resources have been created dynamically even though Erbium does not offer this functionality by default. An initial CoAP POST request has been sent to the IoT controller to start the authentication process. The two were then merged and extended to interact with the EAP state machine to provide a proof-of-concept implementation. Analyzing the traffic during the CoAP-EAP authentication process proved the desired behavior of the IoT device. Beyond that, a second analysis has been done to break down Erbium's behavior with package loss, which is to be expected in real-life examples. Erbium handled retransmitted CoAP POST requests against a no longer existing resource in a way, that they still are allocated with the appropriate resource. The authentication is not impeded by retransmissions.

In step 7 of the CoAP-EAP flow of operation (see figure 2.2) the IoT controller creates OSCORE security context by using a derivative of the exported MSK according to the used EAP method. The IoT device already has all information in step 6 to generate its MSK but waits until it received *EAP Success*. The payload of the message in step 7 contains *EAP Success* to indicate the IoT device, that the MSK has been exported. But this message payload is already encrypted by OSCORE and can not be accessed without having derived the corresponding MSK. Therefore, the implementation of OSCORE in the context of CoAP-EAP needs to be improved in future work.

This work provided a proof-of-concept implementation compliant with IETF draft version 06 of the EAP-based Authentication Service for CoAP. This included a deep dive into the way how Erbium handles web resources and an adaption of Erbium's resource handler functions to create resources dynamically. The test setup consisted of the Cooja Simulator to emulate an IoT device executing the Contiki Operating System with the Erbium REST engine.

Contiki is only one of many operating systems for microcontrollers. Therefore, this implementation is limited to supported hardware by Contiki. However, CoAP implementations [88] also exist for other operating systems like RIOT [89] or Mbed [90]. They might be considered as a base for future work as a foundation to implement the HATEOAS principle not only to Contiki by using Erbium, but to other operating systems as well.



Bibliography

[1] Telecommunication Standardization Sector of ITU, "Overview of the internet of things," *Recommendation ITU-T Y.2060*, 2012. [Online]. Available: https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=11559&lang=en

[2] A. Khan, A. Al-Zahrani, S. Al-Harbi, S. Al-Nashri, and I. A. Khan, "Design of an iot smart home system," in *2018 15th Learning and Technology Conference (L&T)*. IEEE, 2018.

[3] V. Gazis, "A survey of standards for machine-to-machine and the internet of things," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 482–511, 2016.

[4] F. Hüning, *Embedded Systems für IoT*. Springer Berlin Heidelberg, 2019. [Online]. Available: https://doi.org/10.1007/978-3-662-57901-5

[5] M. Sethi, B. Sarikaya, and D. Garcia-Carrillo, "Terminology and processes for initial security setup of iot devices," Working Draft, IETF Secretariat, Internet-Draft draft-irtf-t2trg-secure-bootstrapping-01, October 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-irtf-t2trg-secure-bootstrapping-01.txt

[6] L. F. Rahman, T. Ozcelebi, and J. Lukkien, "Understanding iot systems: a life cycle approach," *Procedia computer science*, vol. 130, pp. 1057–1062, 2018.

[7] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, 2017, pp. 1–7.

[8] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," Internet Requests for Comments, RFC Editor, RFC 7252, 2014. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7252.txt

[9] "Erbium (Er) REST Engine - C CoAP Implementation," Mar. 2018. [Online]. Available: https://web.archive.org/web/20180316172739/http://people.inf.ethz.ch:80/mkovatsc/erbium.php (visited on 2022-04-09).

[10] "contiki-os/contiki Wiki." [Online]. Available: https://github.com/contiki-os/contiki/wiki#Platforms (visited on 2022-04-09).

[11] C. Paar and J. Pelzl, *Understanding cryptography*, 2010th ed. Berlin, Germany: Springer, Nov. 2014.

[12] O. Garcia-Morchon, S. Kumar, and M. Sethi, "Internet of things (iot) security: State of the art and challenges," Internet Requests for Comments, RFC Editor, RFC 8576, 2019.

[13] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler, "{JEDI}: Many-to-many end-to-end encryption and key delegation for iot," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1519–1536.

[14] R. C. Merkle, "Secure communications over insecure channels," *Communications of the ACM*, vol. 21, no. 4, pp. 294–299, 1978. [Online]. Available: https://doi.org/10.1145/359460.359473

[15] U. Blumenthal, F. Maino, and K. McCloghrie, "The advanced encryption standard (aes) cipher algorithm in the snmp user-based security model," Internet Requests for Comments, RFC Editor, RFC 3826, 2004.

[16] L. Shurui, L. Jie, Z. Ru, and W. Cong, "A modified aes algorithm for the platform of smartphone," in *2010 International Conference on Computational Aspects of Social Networks*. IEEE, 2010, pp. 749–752.

[17] "Intel® Data Protection Technology with AES-NI and Secure Key." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard-aes/data-protection-aes-general-technology.html (visited on 2022-05-08).

[18] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz, "Energy analysis of public-key cryptography for wireless sensor networks," in *Third IEEE international conference on pervasive computing and communications*. IEEE, 2005, pp. 324–328.

[19] L. Seitz, S. Gerdes, G. Selander, M. Mani, and S. Kumar, "Use cases for authentication and authorization in constrained environments," Internet Requests for Comments, RFC Editor, RFC 7744, January 2016.

[20] D. Garcia-Carrillo and R. Marin-Lopez, "Lightweight coap-based bootstrapping service for the internet of things," *Sensors*, vol. 16, no. 3, p. 358, 2016.

[21] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2," Internet Requests for Comments, RFC Editor, RFC 6347, January 2012. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6347.txt

[22] G. Selander, J. Mattsson, F. Palombini, and L. Seitz, "Object security for constrained restful environments (oscore)," Internet Requests for Comments, RFC Editor, RFC 8613, July 2019.

[23] I. Stellios, P. Kotzanikolaou, and M. Psarakis, "Advanced persistent threats and zero-day exploits in industrial internet of things," in *Security and Privacy Trends in the Industrial Internet of Things*. Springer, 2019, pp. 47–68.

[24] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[25] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1," Internet Requests for Comments, RFC Editor, RFC 2616, June 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2616.txt

[26] B. Varanasi and M. Bartkov, "Restful spring," in *Spring REST*. Springer, 2022, pp. 45–66.

[27] "What is HATEOAS and why is it important? - The RESTful cookbook." [Online]. Available: https://restcookbook.com/Basics/hateoas/ (visited on 2022-04-14).

[28] J. Postel, "Transmission control protocol," Internet Requests for Comments, RFC Editor, STD 7, September 1981. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[29] B. Pollard, *HTTP/2 in Action*. Simon and Schuster, 2019.

[30] S. R. Jan, F. Khan, F. Ullah, N. Azim, and M. Tahir, "Using coap protocol for resource observation in iot," *International Journal of Emerging Technology in Computer Science & Electronics, ISSN: 0976*, vol. 1353, 2016.

[31] J. Postel, "User datagram protocol," Internet Requests for Comments, RFC Editor, STD 6, August 1980. [Online]. Available: http://www.rfc-editor.org/rfc/rfc768.txt

[32] S. Farrell, "Low-power wide area network (lpwan) overview," Internet Requests for Comments, RFC Editor, RFC 8376, May 2018.

[33] S. K. Sharma, T. E. Bogale, S. Chatzinotas, X. Wang, and L. B. Le, "Physical layer aspects of wireless iot," in *2016 international symposium on wireless communication systems (ISWCS)*. IEEE, 2016, pp. 304–308.

[34] K. Nepomuceno, I. N. de Oliveira, R. R. Aschoff, D. Bezerra, M. S. Ito, W. Melo, D. Sadok, and G. Szabó, "Quic and tcp: a performance evaluation," in *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2018, pp. 00 045–00 051.

[35] C. Bormann, A. P. Castellani, and Z. Shelby, "Coap: An application protocol for billions of tiny internet nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012.

[36] M. Nottingham and E. Hammer-Lahav, "Defining well-known uniform resource identifiers (uris)," Internet Requests for Comments, RFC Editor, RFC 5785, April 2010. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5785.txt

[37] C. Amsuess, Z. Shelby, M. Koster, C. Bormann, and P. V. der Stok, "Core resource directory," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-core-resource-directory-28, March 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-core-resource-directory-28.txt

[38] B. Aboba, J. Malinen, P. Congdon, J. Salowey, and M. Jones, "Radius attributes for ieee 802 networks," Internet Requests for Comments, RFC Editor, RFC 7268, July 2014.

[39] B. Aboba, D. Simon, and P. Eronen, "Extensible authentication protocol (eap) key management framework," Internet Requests for Comments, RFC Editor, RFC 5247, August 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5247.txt

[40] L. Chyrun, L. Chyrun, Y. Kis, and L. Rybak, "Information system for connection to the access point with encryption wpa2 enterprise," in *International Scientific Conference "Intellectual Systems of Decision Making and Problem of Computational Intelligence"*. Springer, 2019, pp. 389–404.

[41] A. DeKok, "Extensible authentication protocol (eap) session-id derivation for eap subscriber identity module (eap-sim), eap authentication and key agreement (eap-aka), and protected eap (peap)," Internet Requests for Comments, RFC Editor, RFC 8940, October 2020.

[42] D. Simon, B. Aboba, and R. Hurst, "The eap-tls authentication protocol," Internet Requests for Comments, RFC Editor, RFC 5216, March 2008. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5216.txt

[43] F. Bersani and H. Tschofenig, "The eap-psk protocol: A pre-shared key extensible authentication protocol (eap) method," Internet Requests for Comments, RFC Editor, RFC 4764, January 2007.

[44] T. Aura, M. Sethi, and A. Peltonen, "Nimble out-of-band authentication for eap (eap-noob)," Internet Requests for Comments, RFC Editor, RFC 9140, December 2021.

[45] B. Aboba and M. Beadles, "The network access identifier," Internet Requests for Comments, RFC Editor, RFC 2486, January 1999.

[46] S. Banerji and R. S. Chowdhury, "On ieee 802.11: wireless lan technology," *arXiv preprint arXiv:1307.2661*, 2013.

[47] G.-V. Jourdan, "Centralized web proxy services: Security and privacy considerations," *IEEE Internet Computing*, vol. 11, no. 6, pp. 46–52, 2007.

[48] R. Marin-Lopez and D. Garcia-Carrillo, "Eap-based authentication service for coap," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-ace-wg-coap-eap-06, December 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-ietf-ace-wg-coap-eap-06.txt

[49] Y. Li, D. Li, W. Cui, and R. Zhang, "Research based on osi model," in *2011 IEEE 3rd International Conference on Communication Software and Networks*. IEEE, 2011, pp. 554–557.

[50] F. F. Yao and Y. L. Yin, "Design and analysis of password-based key derivation functions," in *Cryptographers' Track at the RSA Conference*. Springer, 2005, pp. 245–261.

[51] N. Nikolov, O. Nakov, and D. Gotseva, "Operating systems for iot devices," in *2021 56th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. IEEE, 2021, pp. 41–44.

[52] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *29th annual IEEE international conference on local computer networks*. IEEE, 2004, pp. 455–462.

[53] L. Lopriore, "Memory protection in embedded systems," *Journal of Systems Architecture*, vol. 63, pp. 61–69, 2016.

[54] G. Z. Papadopoulos, A. Gallais, G. Schreiner, E. Jou, and T. Noel, "Thorough iot testbed characterization: From proof-of-concept to repeatable experimentations," *Computer Networks*, vol. 119, pp. 86–101, 2017.

[55] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam, "Architectural issues in java runtime systems," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*. IEEE, 2000, pp. 387–398.

[56] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally, "Internet of things (iot): Research, simulators, and testbeds," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1637–1647, 2017.

[57] K. Roussel, Y.-Q. Song, and O. Zendra, "Using cooja for wsn simulations: some new uses and limits," in *EWSN 2016—NextMote workshop*. Junction Publishing, 2016, pp. 319–324.

[58] "RPL Border Router - Contiki." [Online]. Available: https://anrg.usc.edu/contiki/index.php/RPL_Border_Router (visited on 2022-05-09).

[59] "MSP430F5438 data sheet, product information and support | TI.com." [Online]. Available: https://www.ti.com/product/MSP430F5438 (visited on 2022-04-26).

[60] "contiki/examples/er-rest-example at release-2-7 · contiki-os/contiki." [Online]. Available: https://github.com/contiki-os/contiki/tree/release-2-7/examples/er-rest-example (visited on 2022-04-09).

[61] "Copper (cu) coap user-agent." [Online]. Available: https://github.com/mkovatsc/Copper (visited on 2022-04-09).

[62] "Tunslip utility - Contiki." [Online]. Available: https://anrg.usc.edu/contiki/index.php/RPL_Border_Router#Tunslip_utility (visited on 2022-04-09).

[63] "Browser Extensions - Mozilla | MDN." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions (visited on 2022-04-09).

[64] "An Introduction to Cooja · contiki-os/contiki Wiki." [Online]. Available: https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja (visited on 2022-04-09).

[65] "Ubuntu release cycle." [Online]. Available: https://ubuntu.com/about/release-cycle (visited on 2022-04-09).

[66] "examples/er-rest-example/er-example-client.c at release-2-7 · contiki-os/contiki." [Online]. Available: https://github.com/contiki-os/contiki/blob/release-2-7/examples/er-rest-example/er-example-client.c (visited on 2022-04-09).

[67] "examples/er-rest-example/er-example-server.c at release-2-7 · contiki-os/contiki." [Online]. Available: https://github.com/contiki-os/contiki/blob/release-2-7/examples/er-rest-example/er-example-server.c (visited on 2022-04-09).

[68] "Object-like Macros (The C Preprocessor)." [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/Object-like-Macros.html#Object-like-Macros (visited on 2022-04-11).

[69] "If (The C Preprocessor)." [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/If.html (visited on 2022-04-11).

[70] "apps/erbium/erbium.h at release-2-7 · contiki-os/contiki." [Online]. Available: https://github.com/contiki-os/contiki/blob/release-2-7/apps/erbium/erbium.h (visited on 2022-04-09).

[71] "Concatenation (The C Preprocessor)." [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html#Concatenation (visited on 2022-04-11).

[72] "Macro Expansion (The GNU C Preprocessor Internals)." [Online]. Available: https://gcc.gnu.org/onlinedocs/cppinternals/Macro-Expansion.html (visited on 2022-04-11).

[73] "The GNU C Library - String and Array Utilities." [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_5.html (visited on 2022-04-11).

[74] T. Rothwell and J. Youngman, "The gnu c reference manual," *Free Software Foundation, Inc*, p. 86, 2007.

[75] "contiki/random.h at release-2-7 · contiki-os/contiki." [Online]. Available: https://github.com/contiki-os/contiki/blob/release-2-7/core/lib/random.h (visited on 2022-04-13).

[76] "ascii(7) - Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man7/ascii.7.html (visited on 2022-04-13).

[77] "malloc," publisher: The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition Copyright © 2001-2004 The IEEE and The Open Group. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/functions/malloc.html (visited on 2022-04-14).

[78] "Memory allocation · contiki-os/contiki Wiki." [Online]. Available: https://github.com/contiki-os/contiki/wiki/Memory-allocation (visited on 2022-04-14).

[79] D. A. Alonso, S. Mamagkakis, C. Poucet, M. Peón-Quirós, A. Bartzas, F. Catthoor, and D. Soudris, *Dynamic memory management for embedded systems*. Springer, 2015.

[80] "coap-eap-controller/src/panatiki at master · eduingles/coap-eap-controller." [Online]. Available: https://github.com/eduingles/coap-eap-controller (visited on 2022-05-17).

[81] P. M. Sanchez, R. M. Lopez, and A. F. G. Skarmeta, "Panatiki: a network access control implementation based on pana for iot devices," *Sensors*, vol. 13, no. 11, pp. 14 888–14 917, 2013.

[82] D. G. Carrillo, "Un servicio de bootstrapping basado en coap para redes a gran escala de internet de las cosas," 2019-01-09. [Online]. Available: http://hdl.handle.net/10201/65880

[83] "Tutorial: RAM and ROM usage · simonduq/contiki-ng Wiki." [Online]. Available: https://github.com/simonduq/contiki-ng (visited on 2022-05-10).

[84] "contiki/examples/hello-world at release-2-7 · contiki-os/contiki." [Online]. Available: https://github.com/contiki-os/contiki/tree/release-2-7/examples/hello-world (visited on 2022-05-10).

[85] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, and N. Tsiftes, "The contiki-ng open source operating system for next generation IoT devices," *SoftwareX*, vol. 18, p. 101089, 2022.

[86] "contiki-ng/examples/coap/coap-example-server at master · contiki-ng/contiki-ng." [Online]. Available: https://github.com/contiki-ng/contiki-ng (visited on 2022-05-13).

[87] A. Bhattacharyya, S. Bandyopadhyay, A. Pal, and T. Bose, "Constrained application protocol (coap) option for no server response," Internet Requests for Comments, RFC Editor, RFC 7967, August 2016.

[88] "Constrained application protocol (coap) implementations." [Online]. Available: https://coap.technology/impls.html (visited on 2022-05-13).

[89] "RIOT - The friendly Operating System for the Internet of Things." [Online]. Available: https://www.riot-os.org/ (visited on 2022-05-13).

[90] "Free open source IoT OS and development tools from Arm | Mbed." [Online]. Available: https://os.mbed.com/ (visited on 2022-05-13).

A. Appendix

A.1. Time Schedule

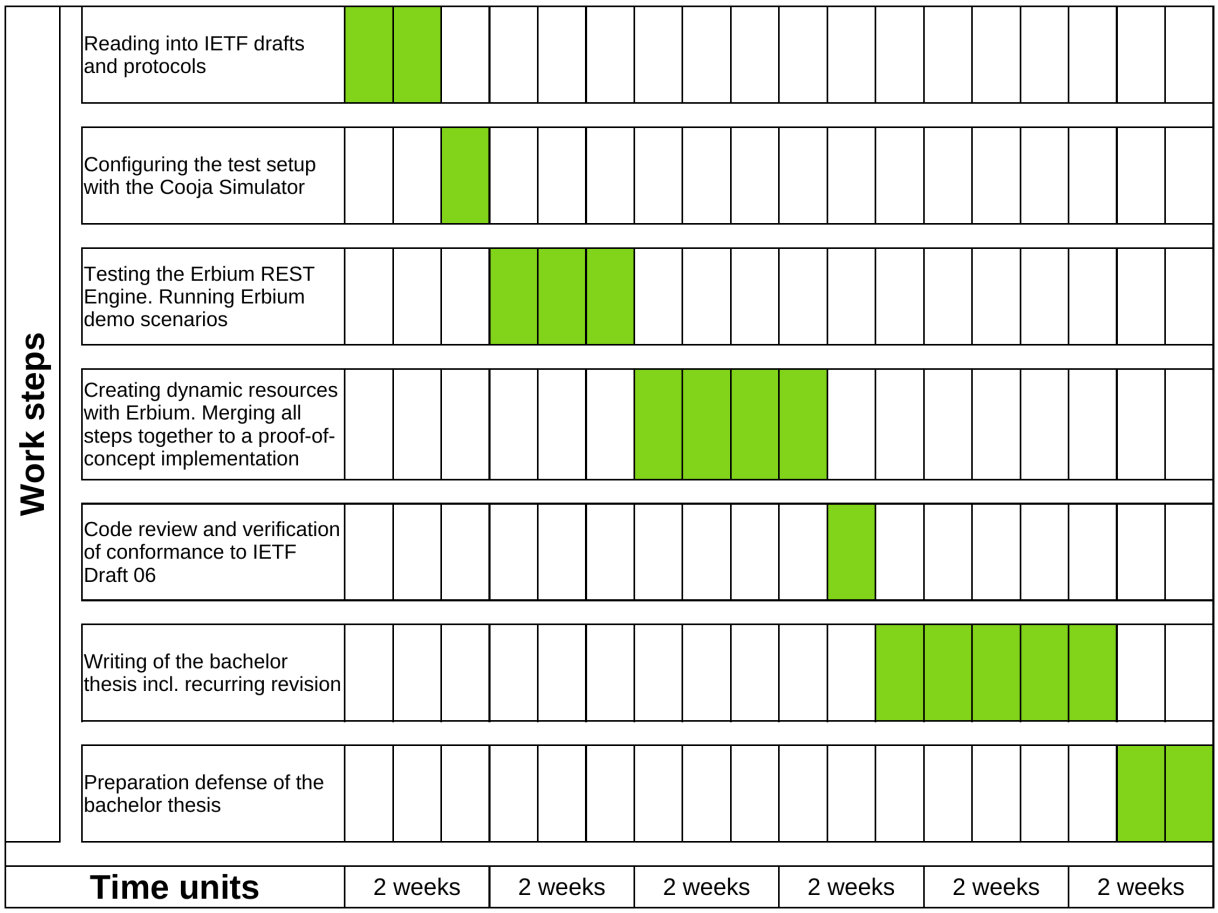


Figure A.1 - Scheduling the individual steps for this thesis

A.2. Source Code of the Proof-of-Concept HATEOAS Implementation on Draft Version 06

Figure A.2 - Source Code of the Proof-of-Concept HATEOAS Implementation on Draft Version 06

The following code shows the content of the file

`Erbium-CoAP-EAP-HATEOAS-IoT-Device.c`.

```
/*
 * Copyright (c) 2013, Matthias Kovatsch
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
```

```
24  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
25  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
26  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
27  * SUCH DAMAGE.
28  *
29  * This file is part of the Contiki operating system.
30  */
31
32 /**
33  * \file
34  *         Erbium (Er) REST Engine example (with CoAP-specific code)
35  * \author
36  *         Matthias Kovatsch <kovatsch@inf.ethz.ch>
37  */
38
39 #include <stdio.h>
40 #include <stdlib.h>
41 #include <string.h>
42
43 #include "contiki.h"
44 #include "contiki-net.h"
45
46 // needs # APPS += eap-sm in the Makefile!
47 #include "eap-peer.h"
48
49 /* Initial resource / hateoas_initial_resource . When called, it will be "overwritten" and a new random Resource will be created. */
50 #define REST_RES_HATEOAS 1
51
52 // Usage of ntohs for the EAP response.
53 #define ntohs(n) (((((unsigned short)(n) & 0xFF)) << 8) | (((unsigned short)(n) & 0xFF00) >> 8))
54
55 // function for debugging the output of the EAP state machine
56 void printf_hex(unsigned char*, int);
```

```
57 void printf_hex(unsigned char* text, int length) {
58     printf("\n");
59     int i;
60     for(i=0; i<length; i++)
61         printf("%02x",text[i]);
62     printf("\n");
63     return;
64 }
65
66 int cipherSuitesSent;
67 uint8_t eapKeyAvailable;
68
69 int counterCryptoSuite = 1;
70 int counterEapResponse = 1;
71 int hateoas_handler_counter = 1;
72
73 /*
74   IP addresses for sending the initial POST request
75 */
76
77 // Examples:
78 // #define SERVER_NODE(ipaddr)   uip_ip6addr(ipaddr, 0xfe80, 0, 0, 0, 0x0212, 0x7402, 0x0002, 0
       x0202) /* cooja2 */
79 // #define SERVER_NODE(ipaddr)   uip_ip6addr(ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0x0001) /* cooja2
       */
80
81 // IP address of the er-example-server mote (the request is sent to the same mote)
82 // #define SERVER_NODE(ipaddr)   uip_ip6addr(ipaddr, 0xaaaa, 0, 0, 0, 0x0212, 0x7402, 0x0002, 0
       x0202) /* cooja2 */
83
84 // IP address for the coap-controller outside of Cooja (see tunslip tunnel)
85 #define SERVER_NODE(ipaddr)   uip_ip6addr(ipaddr, 0xaaaa, 0, 0, 0, 0, 0x00ff, 0xfe00, 0x0001) /*
       cooja2 */
86 #define REMOTE_PORT     UIP_HTONS(COAP_DEFAULT_PORT)
87
88 uip_ipaddr_t server_ipaddr;
89 static struct etimer et;
```

```
90
91  #include "er-coap-13.h"
92  #include "er-coap-13-engine.h"
93
94  /*---------------------------------------------------------------------------*/
95
96  #include "erbium.h"
97
98  /* For CoAP-specific example: not required for normal RESTful Web service. */
99  #if WITH_COAP == 3
100 #include "er-coap-03.h"
101 #elif WITH_COAP == 7
102 #include "er-coap-07.h"
103 #elif WITH_COAP == 12
104 #include "er-coap-12.h"
105 #elif WITH_COAP == 13
106 #include "er-coap-13.h"
107 #else
108 #warning "Erbium example without CoAP-specifc functionality"
109 #endif /* CoAP-specific example */
110
111 #define DEBUG 1
112 #if DEBUG
113 #define PRINTF(...) printf(__VA_ARGS__)
114 #define PRINT6ADDR(addr) PRINTF("[%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x
        :%02x%02x:%02x%02x:%02x%02x]", ((uint8_t *)addr)[0], ((uint8_t *)addr)[1], ((uint8_t *)addr)
        [2], ((uint8_t *)addr)[3], ((uint8_t *)addr)[4], ((uint8_t *)addr)[5], ((uint8_t *)addr)[6], ((uint8_t
        *)addr)[7], ((uint8_t *)addr)[8], ((uint8_t *)addr)[9], ((uint8_t *)addr)[10], ((uint8_t *)addr)[11],
        ((uint8_t *)addr)[12], ((uint8_t *)addr)[13], ((uint8_t *)addr)[14], ((uint8_t *)addr)[15])
115 #define PRINTLLADDR(lladdr) PRINTF("[%02x:%02x:%02x:%02x:%02x:%02x]", (lladdr)->addr
        [0], (lladdr)->addr[1], (lladdr)->addr[2], (lladdr)->addr[3], (lladdr)->addr[4], (lladdr)->addr[5])
116 #else
117 #define PRINTF(...)
118 #define PRINT6ADDR(addr)
119 #define PRINTLLADDR(addr)
120 #endif
121
```

```
122 /*****************************************************************************/
123
124 #if REST_RES_HATEOAS
125 /*
126   The size of the char array can't be modified later.
127   urlString[] will be always (24+1)*sizeof(char) bytes large, if
128   char urlString[] = "hateoas_initial_resource";
129   sizeof(urlString));
130
131   The following line will result in 404 not found!
132   char urlString[] = "/.well-known/a";
133   It has to be
134   char urlString[] = ".well-known/a";
135 */
136 // The following values for the char array urlString[] cause problems with the implementation of
        the CoAP EAP controller.
137 // char urlString[] = "hateoas_initial_resource";
138
139 // Sub-URI work as well
140 // char urlString[] = ".well-known/a";
141
142 char urlString[] = "initHateoasRsc";
143 int urlStringCounter;
144
145   // Skipping first handler call due to retransmission error on coapeapcontroller
146 int skipFirstHandlerCall = 0;
147
148 RESOURCE(hateoas, METHOD_POST, urlString, "title=\"HATEOAS dynamic resource\";rt=\"
        Debug\"");
149
150 void hateoas_handler(void *request, void *response, uint8_t *buffer, uint16_t preferred_size,
        int32_t *offset) {
151   printf("\n");
152   printf("hateoas_handler_counter = %d\n", hateoas_handler_counter);
153   hateoas_handler_counter++;
154   /*
```

```
155  The following will not compile because of the following error: "for loop initial declarations
       are only allowed in C99 mode. Use option -std=c99 or -std=gnu99 to compile your code"
156  for(int i = 0; i < 5; i++) {
157    urlString[i] = 'A' + (random_rand() % 26);
158  }
159  */
160
161  // Skipping first handler call due to retransmission error on coapeapcontroller
162  // Only creating dynamic resources if the eapKey is not available.
163  if(skipFirstHandlerCall == 1 && !eapKeyAvailable) {
164    for(urlStringCounter = 0; urlStringCounter < 5; urlStringCounter++) {
165      // rand typically returns a 16-bit number
166      urlString[urlStringCounter] = 'A' + (random_rand() % 26);
167    }
168    // terminating the string after 5 random uppercase characters
169    urlString[5] = '\0';
170  }
171
172  skipFirstHandlerCall = 1;
173
174  // Following line not needed, because the chars are already altered!
175  // resource_hateoas.url = urlString;
176
177  const uint8_t *payloadData = NULL;
178  int payloadLength = REST.get_request_payload(request, &payloadData);
179  printf("Payload received from the POST request: ");
180  printf_hex(payloadData, payloadLength);
181
182  // variable eapKeyAvailable can not be renamed due to by RFC 4137
183  if(!eapKeyAvailable) {
184        // passing the payload received in the request to the eap_peer_sm_step, see apps\eap-sm\
       eap-peer.c
185        // eapReq from eap-peer.c as well! NECESSARY!!!
186        eapReq = TRUE;
187        eap_peer_sm_step(payloadData);
188        uint16_t len = ntohs( ((struct eap_msg*) eapRespData)->length);
189        // eap state machines reponse is accessible in eapRespData variable.
```

```
190
191         if (cipherSuitesSent == 1) {
192             // return early, because cipherSuites will only sent once
193             // preventing a lot of jumps.
194
195             // see erbium.h in struct rest_implementation_status for the codes
196             REST.set_response_status(response, REST.status.CREATED);
197             REST.set_header_content_type(response, REST.type.TEXT_PLAIN); /* text/plain is
    the default, hence this option could be omitted. */
198             REST.set_header_location(response, resource_hateoas.url);
199
200             // 3rd parameter in set_reponse_payload is size_t length. Is the datatype size_t is
    unsigned integral type. It represents the size of any object in bytes and returned by sizeof
    operator. It is used for array indexing and counting. It can never be negative. The return
    type of strcspn, strlen functions is size_t.
201             REST.set_response_payload(response, eapRespData, len);
202
203             printf("counterEapResponse = %d\n", counterEapResponse);
204             counterEapResponse++;
205
206             printf("eapResponse Data: ");
207             printf_hex(eapRespData, len);
208         }
209
210         else {
211             // sending the ciphersuites to the coap controller, only done once!
212
213             char tempPayload[100] = {0};
214             static char cborcryptosuite[2] = {0x81,0x00};
215
216             memcpy(tempPayload, eapRespData, len);
217             tempPayload[len]=cborcryptosuite[0];
218             tempPayload[len+1]=cborcryptosuite[1];
219
220             // 3rd parameter in set_reponse_payload is size_t length. Is the datatype size_t is
    unsigned integral type. It represents the size of any object in bytes and returned by sizeof
```

```
    operator. It is used for array indexing and counting. It can never be negative. The return
    type of strcspn, strlen functions is size_t .
221             REST.set_response_payload(response, tempPayload, len+2);
222
223             // see erbium.h in struct rest_implementation_status for the codes
224             REST.set_response_status(response, REST.status.CREATED);
225             REST.set_header_content_type(response, REST.type.TEXT_PLAIN); /* text/plain is
    the default, hence this option could be omitted. */
226             REST.set_header_location(response, resource_hateoas.url);
227
228
229             cipherSuitesSent = 1;
230
231             printf("counterCryptoSuite = %d\n", counterCryptoSuite);
232             counterCryptoSuite++;
233
234             printf("eapResponse Data + Ciphersuites: ");
235             printf_hex(tempPayload, len+2);
236         }
237     }
238
239   // executing this until the EAP key is available
240   // Setting OSCORE payload after the eapKey is available
241   else {
242         // Here we would verify the OSCORE Option
243
244         unsigned char oscore_payload[10] = {0x19, 0xf7, 0xcc, 0x6a, 0x15, 0x20, 0x8b, 0x2d, 0xab};
245         unsigned char testzero[1] = {0x00};
246
247         // to do: add a option to the payload which needs to be sent to the controller.
248         // addOption(response,COAP_OPTION_OSCORE, 0, testzero);
249         // setPayload( response, oscore_payload, 9);
250
251         printf("EAP Key is available!\n");
252   }
253 }
254 #endif
```

```
255
256 /*---------------------------------------------------------------------------*/
257
258 // Defines for the EAP state machine
259 #define SEQ_LEN 22
260 #define KEY_LEN 16
261 #define AUTH_LEN 16
262
263 PROCESS(rest_server_example, "Erbium Server with HATEOAS");
264 // PROCESS(coap_client_example, "COAP Client Example");
265 // AUTOSTART_PROCESSES(&rest_server_example, &coap_client_example);
266 AUTOSTART_PROCESSES(&rest_server_example);
267
268 PROCESS_THREAD(rest_server_example, ev, data)
269 {
270   PROCESS_BEGIN();
271
272   // Code for calling the EAP state machine
273   unsigned char auth_key[KEY_LEN] = {0};
274   unsigned char sequence[SEQ_LEN] = {0};
275
276   memset(&msk_key,0, MSK_LENGTH);
277   eapRestart=TRUE;
278   eap_peer_sm_step(NULL);
279
280   cipherSuitesSent = 0;
281
282   memset(&auth_key, 0, AUTH_LEN);
283   memset(&sequence, 0, SEQ_LEN);
284
285   eapKeyAvailable = 0;
286
287   // End of Code for the EAP state machine
288
289   PRINTF("Starting Erbium on the Mote with HATEOAS implementation\n");
290
291   /* Initialize the REST engine. */
```

```
292   rest_init_engine ();
293
294   /* Activate the application-specific resources. */
295
296 #if REST_RES_HATEOAS
297   rest_activate_resource (&resource_hateoas);
298 #endif
299
300 /*****************************************************************************/
301
302     // waiting 15 seconds so that the tunslip tunnel is established and the rest engine loaded
303     etimer_set(&et, 15 * CLOCK_SECOND);
304     int  initial_request_sent  = 0;
305
306     /* Define application-specific events here. */
307
308     while (1)
309     {
310       PROCESS_WAIT_EVENT();
311
312       // Sending the request only once
313       if (etimer_expired(&et) && (initial_request_sent == 0)) {
314           initial_request_sent  = 1;
315         static coap_packet_t initialRequest [1];
316
317         SERVER_NODE(&server_ipaddr);
318
319         coap_init_message(initialRequest, COAP_TYPE_NON, COAP_POST, 0);
320         // sending the first and only request of the IoT device to the CoAP-EAP controllers
    resource /.well-known/coap-eap/
321         coap_set_header_uri_path(initialRequest, "/.well-known/coap-eap");
322         // setting the payload to the resources initial name
323         coap_set_payload(initialRequest, &urlString, sizeof(urlString));
324         // acutally sending the CoAP request
325         COAP_BLOCKING_REQUEST(&server_ipaddr, REMOTE_PORT, initialRequest, NULL);
326     }
327
```

```
328   }     /* while (1) */
329
330   PROCESS_END();
331 }
```