HAW HAMBURG

BACHELOR THESIS
Marcel Stenzel

RACED - A RIOT OS integration of the ACE-OAuth DTLS profile

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Marcel Stenzel

RACED - A RIOT OS integration of the ACE-OAuth DTLS profile

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Informatik Technischer Systeme*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Thomas Schmidt
Supervisor: Prof. Dr. Franz Korf

Submitted on: 07. November 2023

Marcel Stenzel

Thema der Arbeit

RACED - Eine RIOT OS Integration des ACE-OAuth DTLS Profils

Stichworte

RIOT, IoT, Sicherheit, Autorisierung, Authentifizierung

Kurzzusammenfassung

Mit dem Authentication and Authorization for Constrained Environments (ACE) (dt. Authentifizierung und Autorisierung für eingeschränkte Umgebungen) Grundgerüst wird ein System definiert, dass es eingeschränkten IoT Geräten ermöglicht einen Protokollfluss durchzuführen, der Teilnehmer authentifiziert und kontrollierten, autorisierten Zugang zu gesicherten Ressourcen gewährt. Mit einem zusätzlichen Profil wird beschrieben, wie DTLS zur Absicherung der Kommunikation genutzt werden kann. Diese Arbeit implementiert das ACE-Grundgerüst zusammen mit dem genannten Profil für RIOT, einem Betriebssystem spezialisiert auf dieselbe Art von Geräten wie ACE. Diese Implementation besteht zum einen aus einem Modul, dass Funktionalitäten von ACE abstrahiert und es somit Entwicklern zukünftig vereinfacht ihre eigene Implementation von ACE mit RIOT zu produzieren. Zum anderen werden Anwendungen entwickelt, die verbleibende Funktionalitäten, welche nicht abstrahiert werden konnten, beispielhaft darstellen. Außerdem wird die Verwendung des genannten Moduls ebenfalls in diesen Anwendungen dargestellt.

Marcel Stenzel

Title of Thesis

RACED - A RIOT OS integration of the ACE-OAuth DTLS profile

Keywords

RIOT, IoT, Security, Authorization, Authentication

Abstract

The Authentication and Authorization for Constrained Environments (ACE) framework defines a protocol flow that is aimed to enable constrained IoT devices to authenticate participants and authorize controlled access to restricted resources. For this framework a profile is defined that describes the utilization of DTLS to secure the exchanged messages. This work integrates the main protocol interactions of the framework with the mentioned profile into RIOT, an operating system aimed for the same kind of devices as the ACE framework. This integration consists of two parts. The first part is a module that offers functionalities to alleviate work from future developers that want to create their own deployments of the framework with RIOT. The second part is about applications that directly showcase the use of the module, as well as ACE concepts that could not be abstracted into the module.

Contents

List of Figures

List of Tables

Abbreviations

ACE	Authentication and Authorization for Constrained Environments.
API	Application Programming Interface.
AS	Authorization Server.
CBOR	Concise Binary Object Representation.
CoAP	Constrained Application Protocol.
COSE	CBOR Object Signing and Encryption.
credman	(D)TLS Credential Manager.
CWT	CBOR Web Token.
DSM	DTLS Session Management.
DTLS	Datagram Transport Layer Security.
IoT	Internet of Things.
IP	Internet Protocol.
kid	key identifier.
MAC	Message Authentication Code.
OS	Operating System.
PoP	Proof-of-Possession.

PSK	Pre-shared Key.
RACED	RIOT ACE DTLS.
RFC	Request for Comments.
RO	Resource Owner.
RPK	Raw Public Key.
RS	Resource Server.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.
URI	Uniform Resource Identifier.

1 Introduction

1.1 Motivation

Restricting access to resources for specific individuals can have many reasons. Often it is a matter of security or privacy. In a physical way, this restriction could be enforced by a lockable door. Behind that door could be a room that contains appliances to control important infrastructure of the building. Only someone who got the permission to do so should be able to enter.
The same is true for digital devices, with the noteworthy difference that controlling them is not necessarily bound to physical access. These appliances can be embedded systems that are connected to a network and be controlled from anywhere. This makes it important that it is ensured that only the permitted individuals can execute this control. Individuals in this sense can be either people that access the appliances, or it can be other embedded devices such as sensors that regulate the appliances autonomously.
A problem here is that embedded devices are often constrained regarding their hardware. The mechanisms that are needed to fulfill the desire for restricted access are putting additional demand on the available hardware [2].

RFCs 9200 [1] and 9201 [3] describe the Authentication and Authorization for Constrained Environments (ACE) framework. This framework enables constrained devices to conduct a protocol flow which allows the conditional access to resources. Participating entities are identified (authentication) and can act within their approved access rights (authorization).

1.2 Aim of the Thesis

This thesis aims for an implementation of the ACE framework with RIOT OS. Functionalities that can be abstracted for future deployments will form the RACED module.

The remaining mandatory concepts are provided with example applications that guide developers in the creation of their own deployments. The applications developed for this thesis incorporate functionalities provided by the RACED module.

1.3 Thesis Structure

In Chapter 2 background knowledge about security concepts and connected network technologies are explained. Chapter 3 is about the ACE framework, which is implemented in this work.

The next two chapters are about implementation work that was conducted for this thesis. Chapter 4 contains information about work on RIOT modules. This includes additions to existing modules for necessary functionalities, and the creation of a completely new module: RACED. Following that, the implementation of applications for different ACE entities is showcased in Chapter 5.

In Chapter 6 the conducted testing procedure for the implemented work is shown. Followed by evaluations in Chapter 7. The last chapter is about conclusions and an outlook to future work.

2 Background

2.1 Network Security

In this section concepts of network security that play a major role in this thesis are explained.

2.1.1 Confidentiality

Only intended participants should be able to understand a message with contents that are not supposed to be known publicly. If someone else manages to get access to the message it must not be possible to retrieve any critical information from it.
To achieve this, messages can be made unreadable for someone who is not part of the intended audience. With a cryptographic key and an encryption algorithm the sender of a message can create an encrypted message — the ciphertext — out of the unencrypted message — the plaintext. Only someone who is in possession of the corresponding key and algorithm is able to retrieve the plaintext [4, 5].

Symmetric cryptography

In symmetric cryptography, the sender and receiver of a message share a key that is kept secret between the two entities. This key is used by both sides to either encrypt or decrypt messages [4, 5].

While cryptography is a computationally expensive operation, this method has the advantage that it is comparably fast. This holds for the creation of the ciphertext, as well as regaining the plaintext [5].
However, the problem is the distribution of the key among all participants, which needs to be protected from any intruder. [4, 5].

Asymmetric cryptography

For asymmetric cryptography a pair of keys is used. One key is kept private by the owner. The other key is known publicly and can be used by the sender of a message to encrypt it. The receiver, who owns the key pair, can subsequently decrypt the message with the private key [4, 5].
This is possible due to a mathematical relationship between the two keys. However, the relationship is constructed in such a way that it is not feasible to try to derive one key from the other [5].

The asymmetric form of cryptography is slower compared to its symmetric counterpart. But the distribution is no problem since the public key can simply be shared over any unsecured channel. A common method to establish an efficient, secure connection is therefore to utilize asymmetric encryption and share a symmetric key through it. This shared key is applied afterwards for further communication [5].

2.1.2 Integrity

The integrity of a message attests that it was not manipulated on the way to the receiver. In order to prove the integrity, a hash value of the message may be calculated, whereby a hash algorithm turns a variable length message into a fixed length value. This hash value is then appended to the message.

If an intruder intercepts the message in transit, manipulates and then forwards it to the actual receiver, the resulting hash changes. After reception, the receiver calculates the hash as well and compares the value to the one that was sent with the message. In case they are not equal a manipulation is detected. This requires that the hash itself is protected against manipulation. Otherwise, the intruder would be able to simply recalculate the hash and exchange the value. There are two different methods to achieve this protection, the Message Authentication Code and digital signatures. [4, 5].

Message Authentication Code

The Message Authentication Code (MAC) uses a shared key to protect the hash. Now the hash gets calculated with the message itself and the key appended to it. Since only the sender and receiver know the key, an intruder is not able to calculate the correct hash for a manipulated message [4].

Digital signatures

With digital signatures, the integrity of a message can be proven with asymmetric keys. Again, the hash of the message is needed. Except this time there is no shared secret that can be appended for the calculation of the hash. Therefore, the hash needs to be encrypted.
In Section 2.1.1 it was shown that a message can be encrypted with the public key of the receiver. After reception, it is decrypted with the private key. This concept also works the other way around. The sender encrypts the hash with its own private key and appends it to the message. The receiver decrypts the hash with the senders public key and compares the value with the self calculated one [4, 5].
Encrypting the signature with the private key means that everyone, including an intruder, can decrypt and read the signature. But securing the confidentiality of the signature is not the goal of encryption in this case. It is only supposed to prove that the signature, and therefore the contained hash value, originated from the expected sender. Getting the right hash value after decrypting the signature with the public key means that only the owner of the according private key can be the one who encrypted the signature.
Even if an intruder tries to exchange the signature, it would not be possible to generate it in the right way without knowledge of the private key. An attempt to use the public key of the sender for this encryption would mean that the actual receiver is not able to decrypt the signature with the same public key. The key parts of an asymmetric key cannot be used symmetrically. They need the proper counterpart [4].

2.1.3 End-point Authentication

During communication, especially in a secured context, it is often of importance to certify who is on the other end. An intruding entity could just pretend to be someone else. By authenticating the entity it is proven that the claimed identity is legitimate [6, 4, 5].

Information that can be used to authenticate an entity can be divided into different categories. The most important one for this work is the authentication with cryptographic keys, which is something the entity possesses. Besides that, there is also the authentication with something the entity knows, or with biometrics. Examples for these are passwords and fingerprints respectively [6, 5].

Authentication with cryptographic keys is realized by making sure that the other entity is able to use the key that is associated with it. Means to achieve this were specified in Section 2.1.2 in the form of MACs and signatures. For this reason, the message integrity is also known as message authentication.
Using only the integrity mechanisms for authentication can be problematic though. It only proves that the messages were sent at some point by the owner of the keys. An attacker could use old messages, even without knowledge of any key, and replay them. The receiver would get valid messages in the wrong assumption that they originated again from the key owner. To prevent this, a random element, a so-called nonce, is brought in. This nonce, e.g. a random number, differs every time and therefore makes each communication unique. For example, the nonce changes the resulting hash of exchanged messages or other security mechanisms as can be seen later in Section 2.2.2. With this, not only the message but also the end-point can be authenticated [4].

2.1.4 Authorization

In many cases not everyone should have access to everything. Authorization is the process of controlling access requests and deciding who gets access to which resources. For this, specific information about the requesting entity is used to evaluate the rights it contains [5, 6, 1].
In the case of this work, this information is either already known by the evaluating entity or provided through the means of a token. This token is constructed by a third party and secured to make sure the token cannot be manipulated, and the issuer can be authenticated. In case it contains secret information the confidentiality needs to be protected as well [1, 7].

2.2 Related Network Technologies

This work makes use of several technologies. These will be explored in this section.

2.2.1 User Datagram Protocol (UDP)

The User Datagram Protocol is a lightweight transport protocol. It sends the data without establishing a connection beforehand and has only a small header [4] which is defined in the UDP RFC [8].

2.2.2 Datagram Transport Layer Security (DTLS)

UDP is a favorable choice as transport protocol for constrained applications due to its characteristics. But it is missing a way to secure the data transfer.
The Transport Layer Security (TLS) is a well known way to provide authentication between peers as well as confidentiality and integrity for transferred messages [4]. The problem is that TLS needs a reliable connection where message loss and reordering are taken into account.
DTLS [9] builds upon TLS to add security measurements for the unreliable transfer of datagrams. It is designed to be as similar to TLS as possible. However, changes are introduced where necessary to adapt for the use with datagrams.

Before data is exchanged, a handshake is conducted. During this handshake the necessary parameters for an encrypted communication are negotiated. Examples are the algorithm for the encryption or information about keys. This could be an identifier for a shared symmetric key or the public keys of each side. Another important part of the handshake is to exchange nonces.
The keying material together with the nonces are further processed to generate a set of symmetric keys that are unique for one session. Each key has a special purpose and secures either the confidentiality, by encrypting the message, or the integrity, by including it with a MAC. Whereas the keys also differ depending on who is sender and who is receiver of the message [4, 10]. Authentication is achieved by proving the communicated keying material can be used as seen in Section 2.1.2 and 2.1.3. In the case of a symmetric key this is done automatically [4] at the end of the handshake when the first encrypted messages are exchanged. For asymmetric keys the use of a signature is needed which is done at an earlier point of the handshake [10].

2.2.3 Constrained Application Protocol (CoAP)

CoAP is a transfer protocol operating on the application layer of the IoT network stack. It uses a design in which each message is either a request, sent by a client, or a response, sent by a server. A server offers resources that are identified with an Uniform Resource Identifier (URI). These resources can be retrieved, created or changed with specified method codes.
CoAP is specialized for the use with constrained environments and usually relies on datagram transfer like UDP provides it. This helps keeping messages compact and enables asynchronous message exchanges. To secure CoAP on the transport layer, DTLS is the default choice [11].

2.2.4 Concise Binary Object Representation (CBOR)

CBOR serializes data in a binary way. This allows for a compact data representation and makes it suitable for efficient data transfer in constrained applications.
Data is encoded as items that have initial bytes to indicate the type and size of the following bytes which represent the data itself. To group data items CBOR offers encoding for maps and arrays. Additionally, CBOR encoders and decoders are expected to be small in code size which is another advantage for constrained devices [12].

2.2.5 CBOR Object Signing and Encryption (COSE)

COSE utilizes CBOR's compact data format to enable IoT devices to use secure messages with signatures, MACs and encryption. The secured message is hereby serialized with CBOR to provide the payload as well as the metadata that is added by the security mechanism.
Next to that, COSE describes how to encode key objects with CBOR [13].

2.2.6 CBOR Web Token (CWT)

CWTs are used to transport specific security related information called claims between different entities. The claims can be used by the receiving entity to make authorization decisions according to the content of the CWT. Accordingly, a CWT should be secured using COSE to provide authentication and to protect the integrity as well as possibly the confidentiality. CBOR is used for data serialization, whereas the data is encoded within a map. The map keys are pre-defined values so that the receiver can understand the context of the information [14].

2.3 RIOT OS

RIOT is an open source Operating System (OS) with the goal of providing a uniform platform for a variety of constrained devices. As Operating System for the Internet of Things (IoT) it provides the expected features, such as hardware abstraction, and requirements of an OS for constrained devices. The latter include a low memory footprint, energy efficiency, and a network stack fitting for the use cases of such devices [15].

Some of the aforementioned technologies have implementations provided by RIOT. CoAP (*GCoaP*) and the underlying UDP socket come as internal modules, while implementations for DTLS (*TinyDTLS*), CBOR (*NanoCBOR*) and COSE (*libcose*) are imported as packages. This makes it possible to easily incorporate these technologies into the work of this thesis. A socket wrapper for *TinyDTLS*, to offer a RIOT specific API, is integrated as another internal module in the form of *sock_ dtls*.

2.3.1 Credman and DSM

The (D)TLS Credential Manager (credman) and DTLS Session Management (DSM) are two further modules provided in RIOT that are used in this work. Credman holds information about the credentials of an application that can be used for a DTLS session. Whereby credentials consist of the cryptographic key — or key pair — and possibly additional data, such as a key identifier. Both symmetric and asymmetric keys can be used. The credentials are registered to credman with the type of the credential as well as an integer tag. The tag together with the type form a unique identifier for a credential. Credentials registered to credman are used by *sock_ dtls* to conduct the DTLS handshake at the beginning of a connection [16].

DSM saves information about a session, which include the IP address of an endpoint and the state of the session. This module is usually abstracted by *GCoaP*. But RACED introduces a use case which made it necessary to expand DSM and retrieve information from it. This is further explained in Section **??**.

3 The ACE Framework

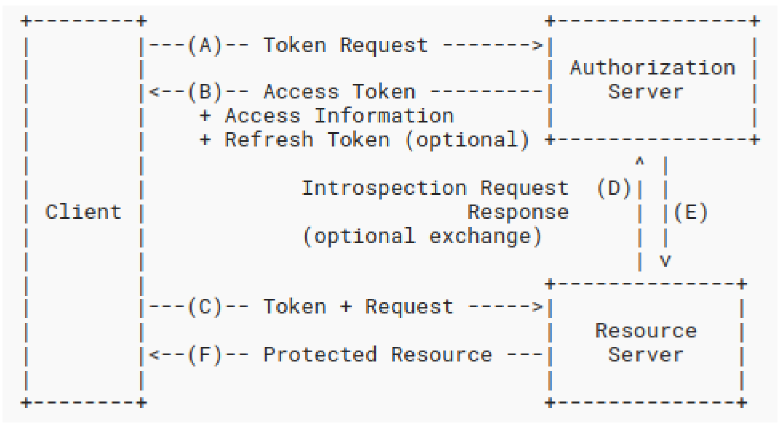
To apply security concepts as they were inspected in Section 2.1, additional resources are needed by an application. This can become a challenge for IoT devices and their often constrained nature. In Section 2.2, technologies that tackle this problem were explained. The ACE framework describes a way to combine these technologies to achieve a packet that fulfills the security concepts.
ACE builds upon OAuth [17], an existing framework for third party authentication and authorization in the web. But in comparison, it tweaks some details and uses technologies to include devices that are limited in their capabilities. Therefore, ACE encourages the use of the technologies mentioned in Section 2.2 instead of the alternatives that are typically used for OAuth. The use of other technologies is not forbidden by the framework, but the technologies mentioned in this work will be the default case [1].

Messages in the protocol flow of this framework are exchanged via CoAP and formatted with CBOR. For formatting, the map functionality is used to encode data with fixed integer keys and connected data types. These key-value-pairs are called claims and are defined in different RFCs. Most of them are defined in the two base RFCs 9200 [1] and 9201 [3]. For the access token, which will be described further in Section 3.2, a CWT formatting is used. Hence, CWT specific map keys are applied in this case. These are defined in RFCs 8392 [14], 8747 [18] and 9200 [1].

3.1 The ACE Entities

There are four different entities that participate in the flow of the ACE framework. As the framework is about restricting access to resources, an entity that offers those resources is needed in the first place. This entity is the Resource Server (RS). Next to that, another role is played by the Resource Owner (RO). The RO decides who is authorized to access the resources. This entity can be either directly involved in the flow or indirectly by

setting up the authorization rules beforehand. Whether the RO is participating directly or not depends on the used grant type, which will be further explored in Section 3.4. Whereas all the other entities are applications running on a device, the RO is an actual person.
An entity that wants to request access to a resource is called the client. For clients, authorization rules are specified that regulate which resources they are allowed to access. The last entity is the Authorization Server (AS). This server holds information that is needed to authenticate a client, capabilities of the client, and possibly the corresponding authorization rules. Whether the authorization rules of a client are deposited within the AS depends on the used grant type. Beyond that, the AS acts as intermediate between the client and the RS for granting the authorization of the resource access. Any RS that this AS is responsible for, is registered with information needed to achieve this role. This information contains details about the offered resources, the capabilities of the RS, and keying material [1]. The interactions between the different entities are shown in Figure 3.1.



```
+--------+                               +---------------+
|        |---(A)-- Token Request ------->|               |
|        |                               | Authorization |
|        |<--(B)-- Access Token ---------|    Server     |
|        |    + Access Information       |               |
|        |    + Refresh Token (optional) +---------------+
|        |                                      ^ |
|        |            Introspection Request  (D)| |
| Client |                         Response     | |(E)
|        |            (optional exchange)       | |
|        |                                      | v
|        |                               +--------------+
|        |---(C)-- Token + Request ----->|              |
|        |                               |   Resource   |
|        |<--(F)-- Protected Resource ---|    Server    |
|        |                               |              |
+--------+                               +--------------+
```

Figure 3.1: Protocol interactions between the different ACE entities without the RO. Source: RFC 9200 [1]

3.2 Protocol Flow

Before a client gets access to a resource, it has to request a token from the Authorization Server (Figure 3.2 step 1). During this step the client and AS authenticate mutually. Further, the request is evaluated according to the authorization rules. In case the client is authorized to access the desired resource, the AS creates an access token and sends it

to the client (Figure 3.2 step 2). Along the token, the response can contain access information that the client needs for setting up a secure communication with the Resource Server.
The token itself is of no use for the client and protected against any manipulation by the client, as it is meant to be forwarded to the RS (Figure 3.2 step 3). An access token is recommended to be a CBOR Web Token, which is one of the security technologies explored in Section 2.2. A CWT is secured with COSE and has therefore a protection for at least the integrity and possibly also for the confidentiality. The necessary protection depends on the keying material that is contained in the token. This can be the public key of the client, a shared symmetric key or even just an identifier if the key itself is known otherwise.
With the token, the RS receives information, such as the keying material, that it needs to set up the secure communication with the client. When the key is used, the client proves that it is in possession of either the same symmetric key or of the private key that is connected to the expected public key. This concept is called Proof-of-Possession (PoP) token and helps to verify that the client has indeed the identity that is connected to the token. Other conveyed information can for example be about the scope of the resources that the token grants access to, for how long the token is valid or with which ACE profile the token can be used.
The client can retrieve all necessary information directly from the access information and does not need the token for it.
Which claims are exactly present in the access token and access information depends on the individual application. Depending on the context and configurations, some information can be known implicitly or are not needed by any of the participating entities. Other parameters such as the used grant type also have an impact on the contents of the exchanged messages [1].

Another important aspect of the COSE security wrapper is that it is used to authenticate the AS and ensure that the token came from a source that is authorized itself to create access tokens for this Resource Server. For this, the RS has to be provided with either a key that it shares with the AS, or with the public key of it. These credentials are used for the CWT protection and therefore validate the security wrapper, i.e. signature, MAC, or encryption [1].

After the RS validated the uploaded token it sends a confirmation back to the client (Figure 3.2 step 4). It is to note that step 3 and 4 can potentially be already part of the setup for a secured channel and are not necessarily separated messages.

Now the client can access resources that it was authorized for (Figure 3.2 step 5 and 6). If any of the requests from a client fails there are specific error responses specified in RFC 9200 [1]. These are illustrated in step 7 and 8 of Figure 3.2 and include cases like a malformed CBOR structure, requesting access to a resource that the client is not authorized for or requesting unsupported parameters [1].

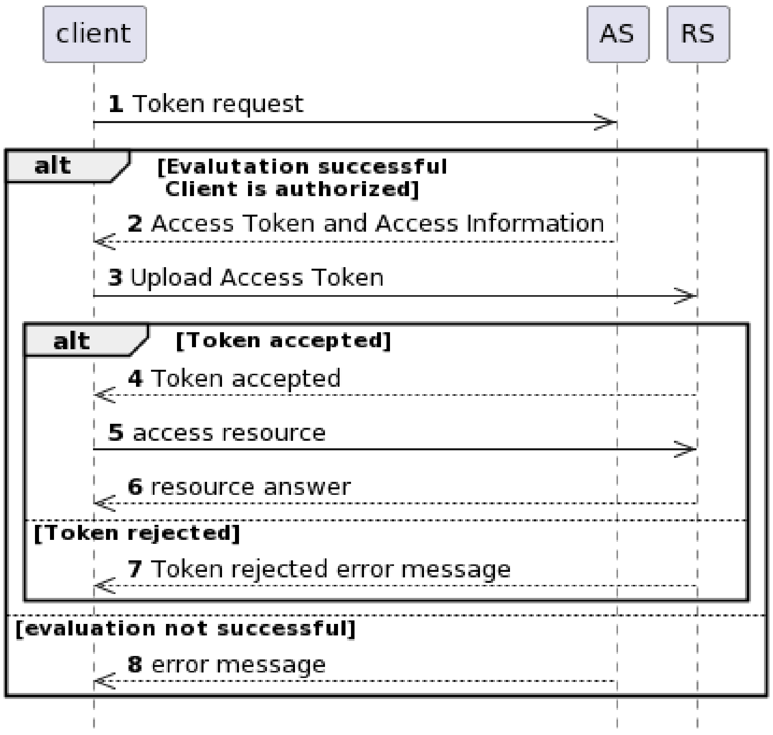


Figure 3.2: ACE protocol flow with error cases

3.3 Profiles

The base ACE RFCs leave some aspects open with regard to securing the communication. Profiles are defined in separate RFCs and are meant to clarify these aspects and expand RFC 9200 with specific transport and communication security protocols [1].

3.3.1 DTLS Profile

RFC 9202 [7] describes a profile that uses the DTLS protocol over CoAP to secure the resource access. The keying material that the AS provides in the access information and access token is therefore applied to set up the corresponding DTLS connection.
A big advantage of DTLS is that it can be used to provide all the security concepts described in Section 2.1 at once. Section 2.2.2 established that protection for the confidentiality, integrity and authentication are naturally incorporated by DTLS. Beyond that, the credentials that were used in the handshake for authentication can also be used for authorization. An access token that the RS receives is bound to the key contained by it. With the DTLS handshake the client fulfills the PoP concept and can therefore be connected to the credentials of the token. Therefore, the authorization that was granted with the token can be applied to the requesting client [7].

One part of the client to RS communication cannot be secured with DTLS, though. Before the RS has knowledge of the necessary keying material to set up the secured channel, it has to receive the access token. This transfer is done over plain, unsecured CoAP. As mentioned before, the access token is a CWT and therefore protected with COSE for this transfer [7].

There are two different modes for this profile that describe if the DTLS connection is set up with either symmetric or asymmetric keys. In Pre-shared Key (PSK) mode the AS provides both the client and the RS with the same symmetric key. As it was explained before, the access token is sent without any transport security to the RS. Since the access token contains a symmetric key, and therefore confidential information, it is mandatory that the token is encrypted.
The second mode uses asymmetric keys in the form of Raw Public Key (RPK). With the access information the client receives the public key of the RS. Further, the RS gets the public key of the client with the access token. This time the token does not need to be encrypted since the public key is no confidential data. However, the token still needs to be integrity protected by a signature or a MAC [7].

How the communication with the AS has to be secured is not predetermined by the RFC, but it is recommended to use DTLS in this case as well. The credentials for this connection have to be distributed in a way that is not further specified [7].

3.3.2 Other Profiles

As this work is focused on the DTLS profile other profiles will only be mentioned briefly.

- RFC 9203 [19] describes a profile that uses application layer security based on COSE for the communication.
- RFC 9430 [20] extends the DTLS profile with the use of TLS.

3.4 Authorization Grant Types

Grant types describe the flow of the communication between the client and the Authorization Server to achieve authentication and granting authorization. ACE uses grant types that were originally defined by OAuth [17]. The default grant type for ACE is the *client credentials* grant, which can be used for pure machine to machine communication. The AS is provided with all the authorization rules and credential information that it needs to make decisions about the authorization without any user interaction. An example for the credentials can be the key that is used for a DTLS connection. Since the client is authenticated with the credentials through the DTLS handshake, the client can be linked to the saved authorization rules. Therefore, the request can be evaluated and authorization can be granted or denied. The implementation of this work will focus on using this grant type.
Other grant types such as the *authorization code grant* directly include the resource owner that can grant authorization through interaction with a device such as a smartphone. RFC 9200 states that most of the use cases of ACE can be accomplished with these two grant types [17, 1].

4 Additions to, and creation of RIOT modules

4.1 Determining Credentials to Authenticate Clients

To implement the ACE framework for RIOT an important aspect needed to be added to the OS. This required changes on the modules for the DTLS socket wrapper and DSM.

4.1.1 The Problem with Determining the Credentials

Prior to this work, it was not possible to determine the credentials that were used in the DTLS handshake on the level of the application. The handling of the credentials in the handshake is conducted fully within the *TinyDTLS* package and its internal socket wrapper. Even though the credentials were registered to credman on application level and the underlying DTLS implementation makes use of these registered credentials, there was no way of knowing which credentials were actually used for the handshake if multiple credentials are registered. This knowledge is important for the ACE framework. A requesting client needs to be further identified by the server, even after a successful handshake. Within the *client credentials* grant type, the client is identified by these credentials. Both server entities need to know exactly which client is sending the request. The authorization server uses this information to connect the request to one of the registered clients and therefore to the rights that were granted to it. Accordingly, the request is evaluated. A similar situation happens within the resource server. It has to connect the request for a resource access to a previously uploaded access token, to know if the access is to be allowed or not.
If a PSK is used by the requesting client, the server knows that the client has at least some form of authorization to request a token or the resource access. Otherwise, the client would not even get past the handshake. If, for example, a resource server only saves one

access token at a time, this might even be enough to properly identify the client. This is an unlikely use case however and would restrict ACE server implementations with RIOT immensely.
For the use of RPK the knowledge of the credentials is even more drastic. In this case only the key pair owned by the server is registered to credman, but not the public key of any client. There is no assurance that the requesting client possesses any authority even if the DTLS handshake is successful. The client only proves to be the owner of the public key used in the handshake. But it is unclear if this public key corresponds to any registered client or access token, until the application has the possibility to check that.

4.1.2 The Implemented Solution

To solve this, the information that DSM holds about a session is expanded with the credman tag and type of the credentials used. In case of asymmetric credentials, additionally the public key of the requesting entity is saved. For this, the DTLS socket wrapper accesses DSM and stores the session information at the time the credentials are determined for the handshake. Like this, the information about the credentials are mapped to the endpoint that is saved in DSM anyway. When a server receives a request it also has information about the endpoint from which the request was sent. Therefore, the credential information can be retrieved from DSM and checked against credentials that are bound to an access token in case of the resource server, or to a registered client in case of the authorization server.

This functionality is separated and not only meant for the RACED implementation, as it can potentially also be used for other use cases. At the moment it is a pending pull request[1] for the RIOT repository to provide it also for any other development that can make use of it.

4.2 The RACED Module

In Section 2.2.4 it was mentioned that CBOR is a good way to serialize data in a compact way that is suitable for constrained environments. The ACE framework makes use of this attribute by defining map constructs to transfer messages of the protocol efficiently. Setting up these maps or extracting the wanted data from it can be a hassle, though.

[1] https://github.com/RIOT-OS/RIOT/pull/19838

As specific map keys have to be inserted, data types have to be taken into account and maps can be nested within other maps.
The RACED module that is implemented for this work achieved to completely abstract any use of CBOR from developers that are implementing their own deployment of ACE with RIOT.
For this, the module offers several data structures as well as functions that take in these structures to en- or decode CBOR data streams. Developers can fill these structures with the data they want encoded or work with already encoded data in an easier way.
Further, the module provides enumeration types of integer abbreviations that are defined by the ACE framework [1]. These help to shorten the messages, for example by providing the used grant type with a single digit. The enumeration on the other hand helps the developer with a written name for the abbreviation.

Even though the thesis focuses on the DTLS profile and the client credentials grant type, the module is valid for other profiles and grant types as well. Solely if new definitions are added by RFCs that describe profiles or grant types these might be missing. But all definitions originating from RFC 9200 [1], 9201 [3] and 9202 [7] are included in the module even if they are not further used in this thesis. Examples for this are the abbreviation for the authorization code grant type and inclusion of claims like username and password in the provided data structures.
This allows for easy extensions of the module as well as serving use cases that go beyond the example implemented for this work.

4.2.1 RACED Data Structures

Data structures are provided for all the different message types within the protocol flow. Therefore, the main structures of the module are for the token request, the token response and the access token. The members of these structure contain all possible claims that can be included in the according CBOR serialization of the message. Since for every claim a member is provided it is important that it is noticeable when a claim is unused. Therefore, the structures should be properly initialized to zero values. For encoding, developers only fill the members that they want to use. Members that have a zero value will be ignored for the encoding. The decoding functions will fill the members that were present in the message. Hence, a member that is still zero after decoding means that the claim was not included. There is one exception to this, the *grant_ type* claim. RFC 9200 [1] defines integer abbreviations for the *grant_ type* claim and zero is a valid value in this case as it stands for the *password* grant type. Instead, the default value that ACE defines for this claim is two — the abbreviation for the *client credentials* grant type.

Other data structures implemented in the module are helping structures that are used within the main structures. *raced_buffer_t* is used to include strings, which can be either a byte string or a text string depending on the claim. The *scope* claim can in fact even be any of those. Therefore, it has its own helping structure called *raced_scope_t* which includes the aforementioned buffer structure as well as a flag that indicates which of the string types it is. This information is important because it changes the type in the resulting data serialization. And the receiving side needs to know which type it is because it can change how the claim is further processed. Mainly it comes down to the fact that the *scope* claim can contain multiple scopes. With a text string it is defined that these are delimited by a *space* character, whereas this is fully undefined for the byte string version [1].
Other helping structures are about the *cnf* claim. This claim can have multiple other claims included and is therefore a map inside the main map. The *cnf* claim is an addition to the CWT and is defined in RFC 8747 [18]. It is used to contain the keying material and has internal claims like a key identifier, an actual symmetric key or information about an asymmetric key.

4.2.2 Structure of the Module

The module is divided into multiple files. Whereby one file is defined here as a .c file together with the corresponding .h file. Figure 4.1 shows an overview of the different parts that the module is composed of.

The Base: RACED

The base file is implemented in *raced.c* and *raced.h* and provides the already mentioned data structures and enumerations. Whereby there are also additional enumerations that are specific to the module and not defined in any RFC. These are *raced_returns_t* for return values of the RACED functions and *raced_cnf_key_type_t*, which is used to indicate what type of key the *cnf* claim holds.
Further, some general help functions are implemented. The first function is *raced_init_-token_request.* Token requests need special initialization since they contain the already

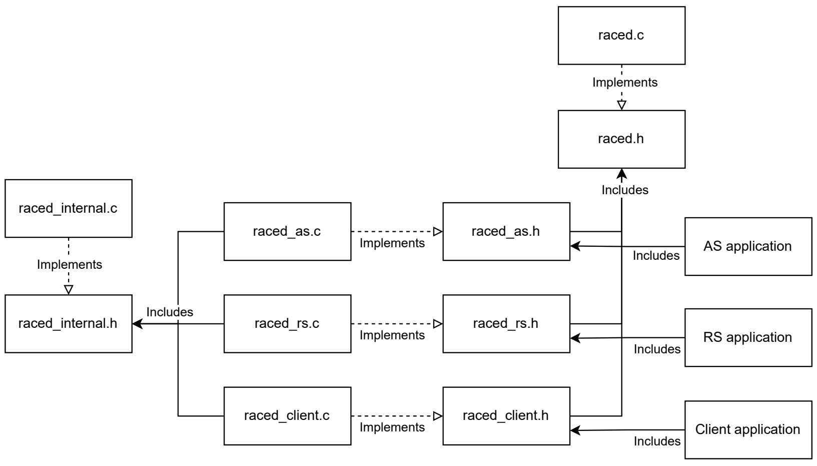


Figure 4.1: Composition of the different parts of the RACED module.

mentioned *grant_type* claim with a default value that is not zero. Secondly, there is the function *raced_parse_psk_kid.* RFC 9202 [7] defines a special structure that the PSK key identifier (kid) needs to have for the DTLS handshake. This function is used to parse a kid to the defined structure.

The ACE Entities

The client, AS and RS parts of the module each have their own files and can therefore be included in a project separately. Each of the files contains functions that are used to encode or decode CBOR data streams corresponding to the needs of the respective entity. *raced_as* for the Authorization Server therefore has the functions *raced_encode_access_-token*, *raced_encode_token_response* and *raced_decode_token_request.* The Resource Server has a corresponding *raced_rs* file with the function *raced_decode_access_token.* Lastly the client file *raced_client* has the functions *raced_encode_token_request* and *raced_decode_token_response.*
Each entity also directly includes the base file, which therefore does not have to be included separately.

Internal Implementations

The last part of the module is *raced_internal.* This provides helping functions for the other RACED files and is not part of the API that is offered to developers. For this

reason functions do not begin with the usual *raced_* prefix.
The functions here avoid code duplication since these routines are used at multiple places, such as taking care of *scope* claims with *encode_scope* and *decode_scope*, which takes into account that the *scope* can be either of the two string types. Next to that, there are functions offered for the *cnf* claim with *encode_cnf* and *decode_cnf*. These additionally make use of some other static functions that manage the different key types that can be included in a *cnf* claim.
Next to the functions this file also holds the CBOR map keys.

4.2.3 Encoding of Data

The functions that encode the provided data structures into CBOR data streams work all in the same principle. After initializing the CBOR encoder, a map is opened and the function goes through all the members of the structure. If a member is present (i.e. not a zero or other default value) the according map key, followed by the value itself with the corresponding indicator for the data type, is written into the buffer for the serialized data. In that regard it is to note that a *grant_type* claim with the value *client_credentials* will not be encoded when using this module. If the claim is missing, the receiving entity has to assume that this grant type is used anyway. Including this value would not transfer any useful information. Therefore, the decision was made for this module to completely leave it out to make messages not bigger than they need to be.
The buffer has to be provided by the application and has to be big enough to hold all the data that is going to be encoded. In case the buffer happens to be too small, the encoding functions return an error value that indicates this problem.
A special case for the encoding is the token response, because this message can transport information about an occurred error. If the *error* claim is set, only this and the other two error related claims, *error_description* and *error_uri*, will be encoded, while all other claims are ignored. This design decision was made for implementations that fill the response structure directly while evaluating the request. In case an error occurred after some members already have been filled, a developer does not need to delete values that are not valid anymore so that only the error members are included.

4.2.4 Decoding of Data Streams

To decode a map encoded CBOR stream of an ACE message, the corresponding functions iterate over the received message and extract the values concealed behind the map keys. Single values, like integers, are directly written into the according structure. Strings on the other hand are only referenced with pointers. This makes it possible to directly access the strings within the serialized data and avoids the need for copying potentially large chunks of bytes that are available in the data stream anyway. The downside is that the developer needs to make sure that the memory that holds the serialized data is still available if these pointers are going to be used at a later time. This design still has advantages as it gives the developer the possibility to only copy necessary parts instead of everything. Further, if the module would copy strings it would be necessary to assign buffers with fixed sizes for every claim that can hold a string, to avoid dynamic memory allocation. This again can result in lots of wasted memory, or in unnecessary restrictions regarding the sizes of those strings.

5 The RACED Applications

The encoding and decoding of the CBOR data streams defined in the ACE framework can be perfectly abstracted and alleviate the development of ACE deployments with RIOT. For other parts of the framework this is not the case since these parts are strongly dependent on the respective application. Therefore, this work implements applications that include the remaining mandatory concepts to showcase a possible deployment of the framework with RIOT and the new RACED module.

In total, three example applications are provided to cover each of the ACE entities.

5.1 Resource Server Application

The RS of this example offers a temperature resource that is represented by an integer value. For this resource the two scopes *read* and *write* are offered and it is available with the URI path */temp-res*. Another CoAP URI is available under */authz-info*. This is the authorization endpoint which is offered for the upload of access tokens as it is described in RFC 9200 [1].
When the RS is being started, the offered CoAP services are registered to the underlying *GCoaP* implementation. With this registration, the URI paths are getting connected to functions that will handle incoming requests. These functions are implemented by the RS itself. This registration also avoids that the CoAP resources can be called with unwanted CoAP methods. Therefore, the URI path are only accessible with the specified methods, which is *post* for the */authz-info* path and *get* along with *post* for the */temp-res* path.

5.1.1 The Token Upload

A token upload to the */authz-info* endpoint will call the *_token_upload_handler* function. This function verifies the validity of the token by conducting a row of checks.
First, the content type of the CoAP message is inspected. This is a value in the CoAP header that indicates that a CWT is included in the message. The next step is the decryption of the token with a symmetric key that the Authorization Server uses to secure the token until its arrival at the RS. Out of simplicity reasons, in the example all access tokens will be encrypted. Even if they hold a public key, and therefore no data that necessarily would need a protection for the confidentiality. In this case a signature could also be used as it was mentioned in 3.3.1. To decrypt the token, functions that are provided by *libcose* are used.
After the token gets successfully decrypted, the included claims are checked. For this, the serialized token is decoded by using the *raced_decode_access_token* function from the RACED module. The resulting representation of the token as a *raced_access_token_t* structure makes it easier to verify the claims. For some claims RFC 9200 [1] defines a list about the priority in which these have to be checked. Therefore, the RS checks in order:

1. The issuer (*iss*) claim, which identifies the AS that generated the token. This identity is authenticated by the COSE wrapper that secured the token.
2. The audience (*aud*) claim, which has to be equal with a value that the RS identifies with.
3. The *scope* claim, which needs to equal to a value that the RS recognizes.
4. The *ace_profile* claim, which needs to be equal to the integer abbreviation of the DTLS profile.
5. The *cnf* claim, which needs to include a COSE key with either an asymmetric or symmetric key.

From this list only the first three are defined with the aforementioned priority. Other claims have no such definitions and can be evaluated in any order. In general, it is not described which claims an access token needs to include. This is dependent on the application, and the claims used in this example were chosen because they are either necessary (*scope* and *cnf*) or for demonstration purpose (*iss*, *aud* and *ace_profile*).

After the uploaded token is fully verified, the RS will store parts of the token for future use. This includes the scope that the token grants access to and the keying material that was conveyed with the token. The other claims were useful for the verification of the token but are not needed any further.
If the type of key that was included in the *cnf* claim was a symmetric key that the RS shares now with the client, this key is getting registered to credman. Like this, the key can be used for setting up the DTLS connection when the client wants to access the resource. The kid that was conveyed with the key needs to be registered to credman in a specific format, as it was described in Section 4.2.2. Therefore, the function *raced_-parse_psk_kid* is used here. In case the *cnf* claim conveyed the public key of the client, this step is not necessary. Only the asymmetric key pair of the RS itself is already registered to credman since the start of the application. The public key of the client is stored independently and is used for authentication at resource access.

Uploaded tokens in this example are handled in a way that the RS stores one access token at a time. When a new token is uploaded it overwrites the current one. A DTLS connection that was set up with the old token is terminated with the acceptance of a new token.

In the end, the RS will send a response with a CoAP message code back to the client. The code is part of the CoAP header and helps the client with indicating how to further process a response. A successful token upload will result in the code for *Created*. A problem with the upload would result in an error code. There are several possible error codes, depending on the occurred problem. Which code is to be used in which case is defined in the ACE RFC 9200 [1]. Therefore, a problem with the claims of a token would result in a *Bad Request* error, a token generated by an AS that is not authorized to do that will be answered with an *Unauthorized* error, and a token that is meant for a different RS results in a *Forbidden* error. Additionally, the example will send an *Internal Server Error* code for problems that are not directly connected to the request of the client. This is not defined in any of the RFCs. An example for such a case is that credentials could not be registered to credman.

5.1.2 The Resource Access

When a client accesses the offered temperature resource, the CoAP handler *_temp_-handler* is called. Depending on the CoAP method that was used for the access a

different granted scope is needed. For the *get* method the scope *read* is needed, and for the *post* method the token needs to grant access for the *write* scope.
The RS implements an internal function called *_guard_resource* that takes the needed scope as an input parameter and conducts checks to verify the validity of the resource access. The first check is about making sure that the access was requested with DTLS. Then, the credentials used for the request are compared to the credentials that are bound to the access token that is currently stored to authenticate the client. For this the newly implemented function, mentioned in Section 4.1.2 is used to retrieve the credentials that were used for the connection. Lastly, the scope needed for the access is compared to the scope granted by the token. When all checks pass, the request continues as a normal CoAP request.
The response that is sent back to the client contains a CoAP message code. Like with the token upload, the codes in error cases are defined in the ACE RFC [1]. Hence, a client that cannot be authenticated or made the request without the use of DTLS will receive an *Unauthorized* code. A request for a part of the resource that is not covered by the granted scope (e.g. requesting to *write* the temperature resource with only the *read* scope is granted) will result in a *Method Not Allowed* code. Requesting access without the granted scope covering that access can also result in a *Forbidden* code. This is the case if the scope only covers completely different resources (i.e. resources with a different URI). Since this example only offers the temperature resource this case is not applicable here. In the same way as mentioned in the token upload, a resource access can also result in an *Internal Server Error* code. The message codes beyond guarding the resource within the protocol flow are entirely specific to the application. This example will answer to a *write* access with the code *Changed*, or with *Bad Request* if the new temperature has more than three digits. A *read* access is answered with the *Content* code and the temperature value appended as payload.

5.2 Authorization Server Application

As it was mentioned in Section 3.3.1, RFC 9202 [7] does not define how the communication between the client and the Authorization Server has to be secured. This example follows the recommendation and uses DTLS for this part of the protocol flow as well. The AS example application is provided with registrations for two different clients. For both clients the AS knows a unique shared symmetric key and also their respective public key. Therefore, clients can authenticate to the AS with either of the key types. Next

to the keying material for authentication, the AS is also provided with knowledge about the scope that each client is able to request. One of the example clients has the right to access just the *read* scope, while the other client is able to access both the *read* and *write* scope at the RS.
Besides the registered clients, the RS is also registered with some information to the AS. This information includes:

- a shared symmetric key for the encryption of access tokens.
- the public key of the RS for inclusion in the access information if a client requests the use of RPK for the connection with the RS.
- the audience that the RS identifies with. This is included in the access token so that the RS has a confirmation that the uploaded token is meant for it.
- the supported ACE profile to confirm compatibility between the client and the RS.

At the start of the AS application, the shared keys with the clients and the asymmetric key pair owned by the AS are registered to credman. The key that it shares with the RS is however not registered to credman, as this key is solely used for the COSE protection wrapper of the access token, and not for a DTLS connection.

5.2.1 Receiving a Token Request

Similar to the RS, the AS registers the CoAP URI for the token endpoint. At this endpoint a client can request a token. It is available with the path */token* and is connected to an internal function called *_token_handler*. Before proceeding with the evaluation of the requests, this function makes sure that the request was received with a secured DTLS connection. Next, the request, which is serialized in a CBOR stream, is decoded into a *raced_token_request_t* structure with the help of the *raced_decode_token_request* function.
The evaluation consists of multiple steps, whereby the first step is to authenticate the client with the credentials that are used in the DTLS connection to send the request. These credentials are retrieved from DSM with the functionality described in Section 4.1.2 and compared to the credentials that are stored with the registered clients. If they fit to one of the clients, the client is identified and the registered information can be used to further evaluate the authorization of this client. After the authentication, the compatibility of the client and the RS regarding the ACE profile is assured. Then, other

parameters sent with the request are evaluated. The *grant_ type* claim needs to be set to *client_ credentials*, or it has to be missing, which results in the same value. If an *audience* claim was present in the request, it has to be equal to the one that the example RS identifies with. In this example this claim is optional, since there is only one RS and therefore the *audience* is also known implicitly. Next, the *scope* claim is verified. The example uses text string scopes. The string of this claim can contain multiple values, delimited by a space character. Therefore, the content of this claim can result in "read" or "read write" for example. The authorization rules for this example are configured in a way that the client needs to be authorized for every requested scope. If the client that is only authorized for the *read* scope requests a "read write" scope the request is fully denied. But other configurations might be thinkable as well. E.g. only granting the *read* part and ignoring the requested *write* scope. Further, the example also implements the *read* scope as a default value if no specific scope was requested. How these configurations are handled is up to the applications implementing the ACE framework and is not defined in any RFC. If the client requests a token that is bound to an asymmetric key it includes the *req_ cnf* claim in the request. This claim contains the public key of the client that it wants to use for communication with the RS. The key needs to be equal to the one that is registered with the client to the AS. Due to the previous authentication, the client proofed the possession of the key.

During the evaluation, the AS fills the structures for the response (*raced_ token_ response_ t*) and the access token (*raced_ access_ token_ t*) at the same time. Whereby, the response is addressed to the client and the access token is addressed to the RS. Both is sent to the client, and it is the clients task to forward the token to the RS. The response is filled with the following claims:

- *ace_ profile* set to the integer abbreviation of *coap_ dtls*, if the client sent an empty *ace_ profile* claim in the request.
- *scope*, containing the scope that the client is granted access to. This is either the exact scope the client requested or the default value if no scope was specified in the request.
- *rs_ cnf*, with the public key of the RS. This claim is only included if the client requested the use of asymmetric keys.

- *cnf*, with a symmetric key and kid that the AS generated for the communication between client and RS. This claim is only included if the client did not explicitly ask for the use of asymmetric keys.

The access token is filled with the following claims:

- *aud*, containing the audience that the RS identifies with that this token is created for.
- *scope*, with the same value that is also included in the *scope* claim of the response.
- *cnf*, with either the public key of the client or the same symmetric key material as included in the *cnf* claim of the response. The content depends on the request.
- *iss*, with the identifier of the AS.

For the generation of the symmetric key the *cose_crypto_keygen* function from *libcose* is used. The kid is generated with the help of another RIOT module called *LUID*, which can be used to generate locally unique identifiers for different purposes.

If the evaluation fails, an according error, defined in RFC 9200 [1], is set for the *error* claim of the response. This can happen for example because the client requests a scope it is not authorized for, or the request comes from a client that is not registered at the AS. In this case only the error claims will be encoded later on, as it was described in Section 4.2.3.

After successful evaluation of the request, the access token is encoded into a CBOR serialization with the *raced_encode_access_token* function and afterwards encrypted with COSE. The result is then included as a byte string in the response with the *access_-token* claim. Now that the response is complete, it can be encoded with the *raced_-encode_token_response*.
Finally, the response is sent back to the client via CoAP, along with a CoAP message code that indicates if the request was successful (*Created* code), or if there was a problem with it (*Bad Request* code). The message codes which are to be used are defined in RFC 9200 [1]. As it was described before for the RS example in Section 5.1 the *Internal Server Error* code can be used here too.

5.3 Client Application

There are two different credential settings which can be used with the client example. These correspond to the client registrations that were mentioned before with the AS example in Section 5.2. Hence, one setting represents a client with rights for the *read* access, while the other setting is for a more privileged client with access rights for both the *read* and *write* scope. Which credentials are taken can be chosen with the constant *USE_PRIVCLIENT* at the beginning of the .c file of the client example. If the constant equals zero, the client with only read access is taken. Otherwise, the credentials for the privileged client are used.
During the initialization, the client application registers the symmetric key that it shares with the AS, as well as its own asymmetric key pair to credman for upcoming DTLS connections. Which credentials are registered depends on the configuration in the Makefile of the client example. Only the keying material for activated functionalities is registered. These functionalities are activated by including *CFLAGS += -DCONFIG_DTLS_PSK* for symmetric keys or *CFLAGS += -DCONFIG_DTLS_ECC* for asymmetric keys. It is to note that the underlying *TinyDTLS* implementation will favor the choice of the asymmetric key for the handshake if both options are activated at the same time. The application cannot influence this choice. Hence, it is necessary to compile the client application without DTLS support of asymmetric keys if the authentication with a symmetric key is wanted. For a mixed authentication that uses RPK with the AS and subsequent PSK with the RS, this configuration has to be made on the Makefile of the RS. Like this, the use of a symmetric key with the RS can be enforced, even if both modes are enabled at the client as the use of asymmetric keys cannot be negotiated during the DTLS handshake. Otherwise, the client will automatically use its public key for the handshake and the RS will not be able to authenticate the client with the PSK that is bound to the token. This restriction can be problematic for real deployments of the framework with RIOT since the different entities can be developed by entirely different organizations and such configurations are out of reach.

The client offers multiple commands with which requests can be sent to either the AS or the RS. These commands are about requesting a token, uploading the token, and accessing a resource.

5.3.1 Requesting a Token

For the token request a command is implemented with the function *token_request_cmd*. This function takes the address of the AS as an input. The CoAP settings are regulated internally. As such the CoAP header is prepared with the expected method (*Post*), the right format (*ace_cbor*) and the payload. The claims, which are to be included in the request are set within the function and filled into a *raced_token_request_t* structure. Before the actual claims are filled in, the structure is initialized with the *raced_init_-token_request* function. This ensures that unused claims will be set to the default value, especially the *grant_type*, which has a default value that is not zero. After inserting the claims, the request structure is encoded into a CBOR stream with the *raced_encode_-token_request* function and finally sent via DTLS to the provided address. The transfer of the request is entirely managed by the underlying CoAP and DTLS implementations with the *gcoap_req_send_tl* function.

In addition to handing over the message, this function also registers a callback handler for the expected response. This handler is implemented by the client application with the function *_token_resp_handler*. This function analyzes the CoAP code of the response, which indicates if the request was a success, or if an error occurred. If the error code indicates that there was a problem with the request itself (i.e. a *Bad Request*), an encoded payload with at least the *error* claim is expected. For other error codes, the code is printed to the console and the function finishes. A successful response and a *Bad Request* response will continue with the decoding of the serialized response. This is done with the help of the *raced_decode_token_response* function. The claims of the response are processed afterwards with the resulting *raced_token_response_t* structure.
For a *Bad Request*, the *error* claim and, if present, the *error_description* and *error_uri* are printed to the console. This provides the user with the occurred ACE error code and potential further information.
In case of a successful request, the *ace_profile* claim is checked, if present. This claim should contain the integer abbreviation for the DTLS profile. The other two claims in the response are about the keying material. Either the response contains a *cnf* claim with symmetric keying material, or a *rs_cnf* claim with the public key of the RS. This public key can be used to authenticate the RS on application level. The symmetric key and the kid are registered to credman for the future resource access at the RS. Whereby the kid is parsed into the format as described in Section 4.2.2 with the *raced_parse_psk_kid* function.

5.3.2 Uploading a Token

The access token, which was received and stored with the response from the AS, is meant to be forwarded to the RS. For this part of the protocol flow, another command is implemented for the client application. It is available in the console with *upload_-token* and implemented with the function *upload_token_cmd*. As argument, it takes the address of the RS. The CoAP header settings are handled internally. The message is set up with the *Post* method, the */authz-info* path and a format type that indicates that the message contains a CWT. Finally, the access token, which was conveyed with the *access_token* claim in the response, is appended as payload to the message.
This message is sent to the RS with plain CoAP. As mentioned before in Section 3.3.1, this exchange is not secured by DTLS, since the RS is only now receiving the necessary keying material to authenticate the client. Instead, the access token itself is secured with a COSE wrapper.
Again, the message is handed over to the underlying CoAP implementation and a handler for the response is registered. This handler is implemented with the function *_rs_-resp_handler*. As there is no payload expected in the response for the token upload, this handler will simply print out the returned CoAP message code to the console.

5.3.3 Accessing a Resource

The resource access is split into two different commands. The first command for a *read* access is available in the console with *read_res* and is implemented in the example with the function *read_res_cmd*. The address of the RS has to be provided as an argument. The CoAP message is set up automatically with a *Get* method and the destination URI */temp-res*.
For the *write* access the command is *write_res*, which is implemented with the function *write_res_cmd*. Like before, this command takes the address of the RS as input and an additional argument for the temperature value that is going to be written to the resource. The CoAP method Post, the URI */temp-res* and the format type *Text* are set automatically.
In both cases the message is handed over to *GCoaP* and sent to the provided address with DTLS. The response handler is the same as for the token upload. This handler will print the message code of the response, and in case of a *read* access also the received payload.

5.4 Peculiarities with the PSK Handshake

The DTLS handshake is entirely taken care of by the underlying *TinyDTLS* implementation, which is overall very useful since an ACE application does not need to do implement DTLS related behavior itself. However, in PSK mode the inability to influence the handshake introduces some restrictions.

5.4.1 Invalid Client

An invalid client without any authorization is supposed to get a serialized response from the AS with error code *invalid_ client* in the *error* claim. In PSK mode this is not possible since the client will use a symmetric key that is not registered to credman. Therefore, the client will be rejected before the AS application has a chance to check the credentials itself and construct the according response structure.
In RPK mode this is not a problem since the client will pass the DTLS handshake with any asymmetric key pair. The server application will then conduct the authorization check by comparing the stored credentials with the credentials used for the authentication during the handshake, which are retrieved from DSM. This enables the server to construct the expected CBOR answer.

5.4.2 Key ID Structure

During the DTLS handshake in PSK mode, the two peers exchange the key identifier for the key they want to use for the DTLS session. As mentioned in Section 4.2.2, RFC 9202 [7] defines a specific CBOR structure in which the kid needs to be set up for this exchange. Because of the missing possibility to influence the handshake, the kid cannot be extracted from the structure. Therefore, the kid needs to be registered to credman with the whole serialized structure. In that way *TinyDTLS* can use it as a whole as identifier to compare it with the incoming kid.
Since the structure is set up with maps there are two options of what the serialization could look like. One is with map entries that already contain the number of items inside the map. The other one is with indefinite maps that are terminated with a specific value at the end. The example in RFC 9202 [7] uses the first option. This option is three bytes shorter since the terminating values are missing and the structure contains three maps. The content of the structure is, with exception of the actual key identifier, always build

up in the same way and the number of map entries is known. For these reasons, the implementation of this work saves the identifier in the way of the first option.
Because of the missing possibility to evaluate incoming identifiers in the handshake, any other implementation that this work wants to interoperate with needs to handle it the same way. Otherwise, the byte representation of the kid in the handshake and the representation of the saved identifier will not be the equal. That would lead to an unknown identifier and the connection establishment is rejected, even if the actual identifier itself would be correct.

5.4.3 Token Inclusion in the Handshake

RFC 9202 [7] describes another method to make the token known to the RS. This method includes the token directly in the DTLS handshake instead of uploading it beforehand. For the same reason as before, this method is not possible when using RIOT to deploy an ACE RS. The application has no access to the token included in the handshake and can therefore not evaluate it.

6 Testing

6.1 Unit Tests

RIOT includes a testing framework called *embUnit* that allows to write unit tests for modules. This helps to confirm a proper functionality, as well as challenging the robustness of the module. To provide a reliable software, this work also provides unit tests for the RACED module.
Tests are expected to be written for every .h file that the module exposes, and to have a specific naming pattern. Therefore, the tests are divided into the files *tests-raced-raced.c*, *tests-raced-raced_as.c*, *tests-raced-raced_rs.c*, *tests-raced-raced_client.c* and *tests-raced-raced_internal.c*. Additionally there are two files that are needed for the *embUnit* set up: *tests-raced.c* and *tests-raced.h*.
The unit test cases provided for the RACED module are shown in Table 6.1, 6.2, 6.3, 6.4 and 6.5. Whereby the first column shows the abbreviated name of the respective test functions without the *test_raced_* prefix.
The unit tests provide special value for functionalities that are not used in the example application, since these are not covered by any manual interoperability testing.

Abbreviated test function	Tested behavior
init_to-ken_request	This confirms a properly initiated *raced_token_request_t* structure with the *grant_type* claim set to *client_credentials* and otherwise zero values.
parse_psk_-kid	This tests the successful parsing of a kid as it was described in 4.2.2.
parse_psk_-kid_nomem	Here, a buffer that is too small is provided for the parsing of a kid. This expects an error return.

Table 6.1: Unit tests for raced.h

Abbreviated test function	Tested behavior
encode_access_token	This confirms the successful encoding of an access token. First, with no value for any claim to test missing claims while encoding. Second, with a value for every claim.
encode_access_token_nomem	This provides the encoding of an access token with a buffer that is too small and expects an error return.
encode_token_response	This confirms the successful encoding of a token response. First, with no value for any claim, to test missing claims while encoding. Second, every claim except for the *error* claim is filled. Like this all claims except for the three error claims, as described in 4.2.3, should be encoded. After that, the *error* claim is filled for the next test part. Now, only the three error claims should be encoded.
encode_token_response_nomem	This tests the encoding of a token response with a provided buffer that is too small. An error return is expected.
decode_token_request	Here, the successful decoding of a token request is confirmed. First with a request that contains no claims. The corresponding *raced_token_request_t* structure should be set to zero values with exception of the *grant_type* claim, which should contain the *client_credentials* grant type. The second test part is about decoding a token request with every claim filled.
decode_token_request_malformed	This tests the decoding of a token request with different errors. The error cases are: an unknown map key, a malformed claim, the request is not formatted as a CBOR map, and the map key has a wrong datatype. This expects error returns for all cases.

Table 6.2: Unit tests for raced_as.h

Abbreviated test function	Tested behavior
decode_access_token	This confirms the successful decoding of an access token. First, with an empty access token to confirm proper behavior with missing claims. The second test part has every claim filled with a value. This confirms that every member of the *raced_access_token_t* structure is getting filled as expected.
decode_access_token_malformed	This decodes access tokens with different errors included. The error cases are the same as for the *decode_token_request_malformed* test.

Table 6.3: Unit tests for raced_rs.h

Abbreviated test function	Tested behavior
encode_token_request	First, a token request with *grant_type* claim set to *client_credentials* and the rest of the claims set to a zero value is encoded. This confirms proper behavior with the default values since this should result in an empty CBOR map. The second part of the test fills a *raced_token_request_t* structure with all members set to a value that is supposed to show up in the serialized data.
encode_token_request_nomem	This test tries to encode a token request with a buffer that is too small and expects an error return.
decode_token_response	Here, the successful decoding of a token response is confirmed. First, with all claims missing, and second, with all claims set.
decode_token_response_malformed	This tests the decoding of a token response with different errors that are equal to *decode_token_request_malformed.*

Table 6.4: Unit tests for raced_client.h

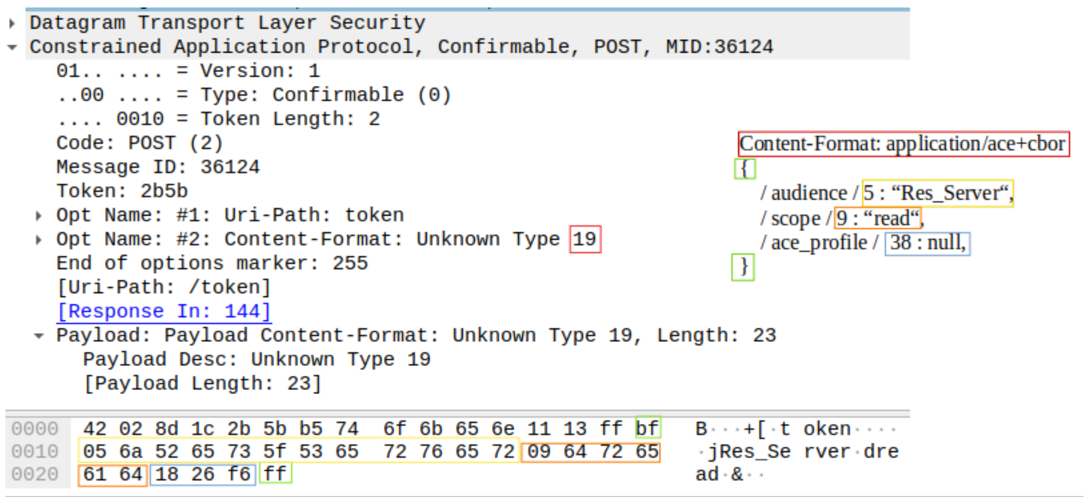
Abbreviated test function	Tested behavior
encode_cnf	This tests the successful encoding of a *cnf* claim with multiple settings. First, with an empty COSE key, where only the key type claim is set, as this is a mandatory claim. The second part fills the empty members with information for an asymmetric key. As third part, a COSE key with a symmetric key is tested. Two more test parts are conducted for the remaining types of a *cnf* claim. Therefore, the claim gets filled with an encrypted key and with only a key identifier respectively.
encode_-cnf_errors	Here, the encoding of a *cnf* claim is used with different errors included. The first test part has no type for the *raced_cnf_t* structure set, which is needed for the function to determine if a COSE key structure, an encrypted key or just a key identifier is to be encoded. The second test part tries to encode a COSE key without having the key type claim set. And the last part provides a buffer that is too small.
decode_cnf	The test parts for successful decoding of a serialized *cnf* claim are similar to the ones of the *encode_cnf* test.
decode_-cnf_mal-formed	For this test, the decoding of a serialized *cnf* claim is called with different errors. These errors are: the claim is not formatted as a CBOR map, the map key has a wrong datatype or an unknown value, a malformed claim, the map does not contain any information, and a COSE key is included without a specified key type. This test expects error returns for the different issues.
encode_-scope	As the *scope* claim can be either a text string or a byte string, this tests the encoding with both possibilities.
encode_-scope_-nomem	This test tries to encode a *scope* claim with a buffer that is too small and expects the appropriate error as return.
decode_-scope	Here, the decoding of a *scope* claim with both possible string types is tested.
decode_-scope_mal-formed	This tests the *scope* claim decoding with malformed data.

Table 6.5: Unit tests for raced_internal.h

6.2 Functionality Test

The implemented example applications are able to communicate with each other. With the command *ifconfig*, which is provided by RIOT, the network address can be retrieved and used for the commands implemented by the applications.

Figure 6.1 shows a token request to get a token bound to a PSK with some other parameters as test. The resulting token response can be seen in Figure 6.2. To get this response, the client authenticated with a PSK that it shares with the AS. With the response, the AS authorizes the client to access the requested *read* scope at the desired audience. The response includes the generated keying material, information about the granted access and an encrypted token.



```
▸ Datagram Transport Layer Security
▾ Constrained Application Protocol, Confirmable, POST, MID:36124
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0010 = Token Length: 2
    Code: POST (2)
    Message ID: 36124
    Token: 2b5b
  ▸ Opt Name: #1: Uri-Path: token
  ▸ Opt Name: #2: Content-Format: Unknown Type 19
    End of options marker: 255
    [Uri-Path: /token]
    [Response In: 144]
  ▾ Payload: Payload Content-Format: Unknown Type 19, Length: 23
      Payload Desc: Unknown Type 19
      [Payload Length: 23]

0000  42 02 8d 1c 2b 5b b5 74  6f 6b 65 6e 11 13 ff bf   B···+[·t oken····
0010  05 6a 52 65 73 5f 53 65  72 76 65 72 09 64 72 65   ·jRes_Se rver·dre
0020  61 64 18 26 f6 ff                                  ad·&··
```

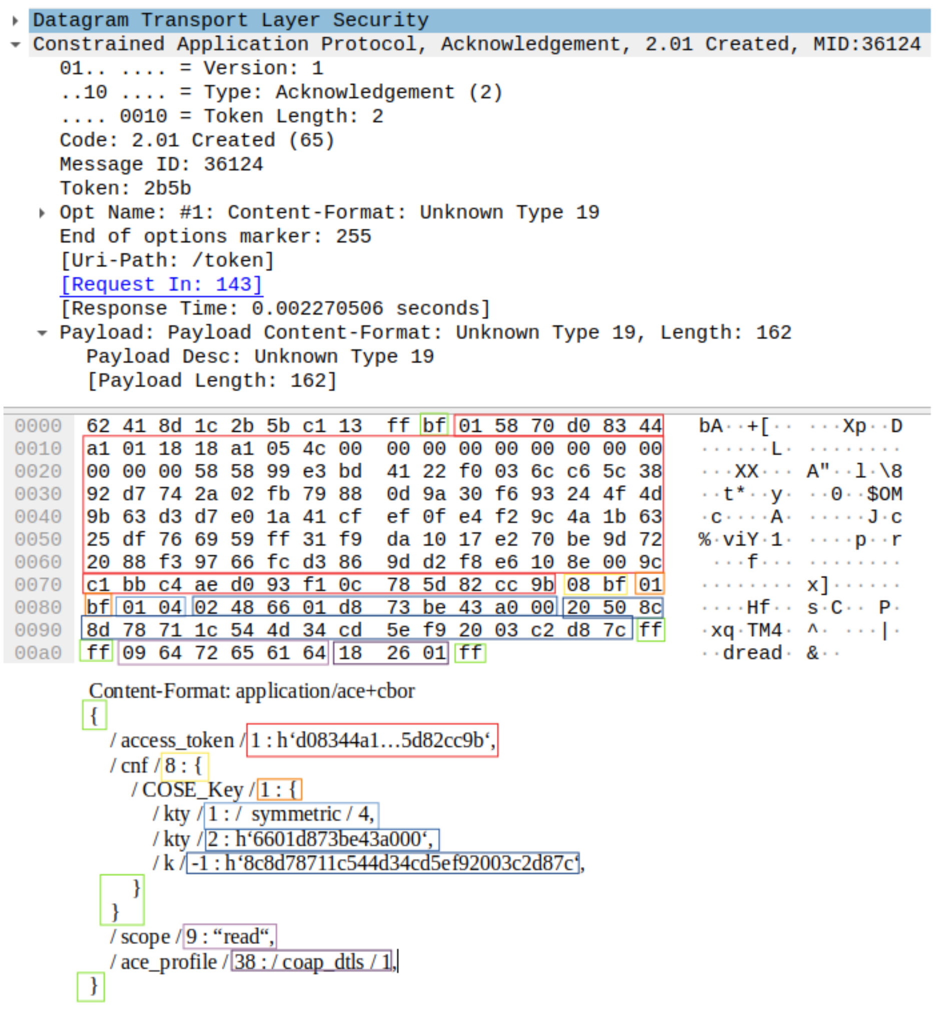
```
Content-Format: application/ace+cbor
{
  / audience / 5 : "Res_Server",
  / scope / 9 : "read",
  / ace_profile / 38 : null,
}
```

Figure 6.1: A token request with the provided parameters on the right side and the resulting message captured with Wireshark.

Next, the client uploads the token that it received from the AS to the RS. The RS evaluates the token and sends an acknowledgment back to the client to confirm the upload (see Figure 6.3).

After the upload, the client can request access to the desired resource. With the keying material and the information about the authorization of the client, the RS can evaluate and grant the requested access as shown in Figure 6.4.

A test to request a token bound to a RPK is not shown as it is very similar. In this case the token request contains an additional *req_ cnf* claim with the public key of the client.



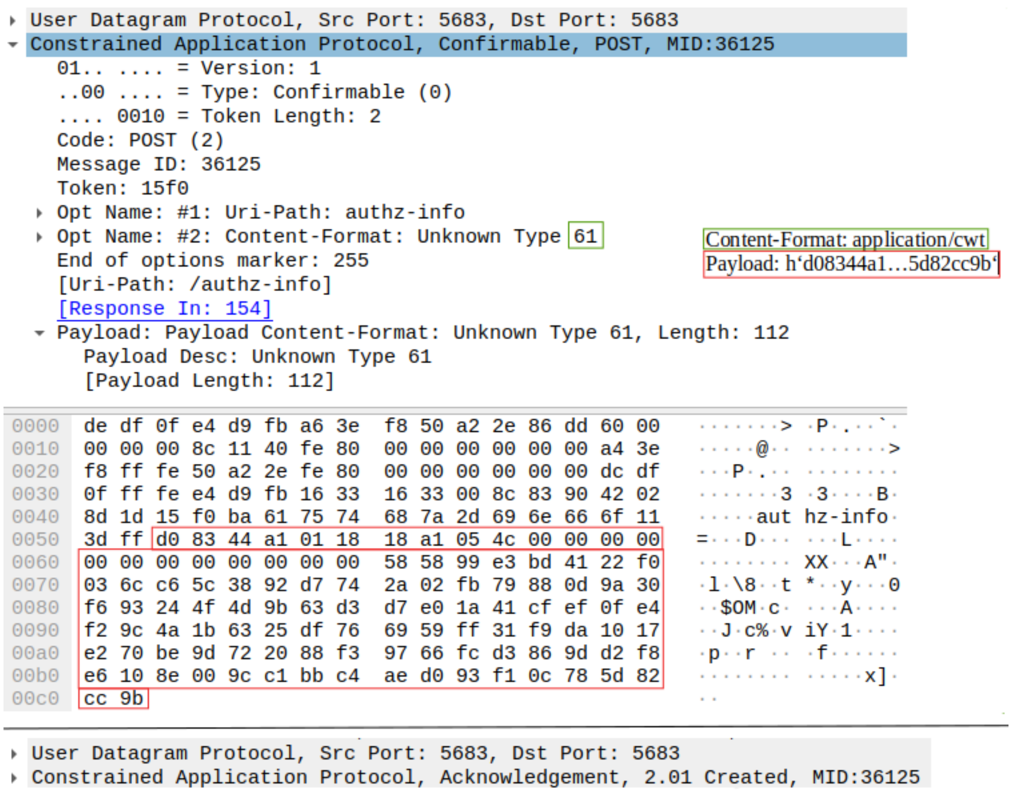
```
▸ Datagram Transport Layer Security
▾ Constrained Application Protocol, Acknowledgement, 2.01 Created, MID:36124
    01.. .... = Version: 1
    ..10 .... = Type: Acknowledgement (2)
    .... 0010 = Token Length: 2
    Code: 2.01 Created (65)
    Message ID: 36124
    Token: 2b5b
  ▸ Opt Name: #1: Content-Format: Unknown Type 19
    End of options marker: 255
    [Uri-Path: /token]
    [Request In: 143]
    [Response Time: 0.002270506 seconds]
  ▾ Payload: Payload Content-Format: Unknown Type 19, Length: 162
      Payload Desc: Unknown Type 19
      [Payload Length: 162]

0000  62 41 8d 1c 2b 5b c1 13  ff bf 01 58 70 d0 83 44   bA··+[··  ··Xp··D
0010  a1 01 18 18 a1 05 4c 00  00 00 00 00 00 00 00 00   ······L·  ········
0020  00 00 00 58 58 99 e3 bd  41 22 f0 03 6c c6 5c 38   ···XX···  A"··l·\8
0030  92 d7 74 2a 02 fb 79 88  0d 9a 30 f6 93 24 4f 4d   ··t*··y·  ··0··$OM
0040  9b 63 d3 d7 e0 1a 41 cf  ef 0f e4 f2 9c 4a 1b 63   ·c····A·  ·····J·c
0050  25 df 76 69 59 ff 31 f9  da 10 17 e2 70 be 9d 72   %·viY·1·  ····p··r
0060  20 88 f3 97 66 fc d3 86  9d d2 f8 e6 10 8e 00 9c    ···f···  ········
0070  c1 bb c4 ae d0 93 f1 0c  78 5d 82 cc 9b 08 bf 01   ········  x]······
0080  bf 01 04 02 48 66 01 d8  73 be 43 a0 00 20 50 8c   ····Hf··  s·C·· P·
0090  8d 78 71 1c 54 4d 34 cd  5e f9 20 03 c2 d8 7c ff   ·xq·TM4·  ^·  ···|·
00a0  ff 09 64 72 65 61 64 18  26 01 ff                  ··dread·  &··
```

```
Content-Format: application/ace+cbor
{
   / access_token / 1 : h‘d08344a1…5d82cc9b‘,
   / cnf / 8 : {
      / COSE_Key / 1 : {
         / kty / 1 : /  symmetric / 4,
         / kty / 2 : h‘6601d873be43a000‘,
         / k / -1 : h‘8c8d78711c544d34cd5ef92003c2d87c‘,
      }
   }
   / scope / 9 : “read“,
   / ace_profile / 38 : / coap_dtls / 1,
}
```

Figure 6.2: Response to the token request with the message captured with Wireshark and the encoded claims below.

In the response, the same public key is included in the *cnf* claim of the access token. Further, the *cnf* claim of the authorization information is exchanged with a *rs_ cnf* claim that includes the public key of the RS. The proper functionality with asymmetric keys is proven in Section 7.1.

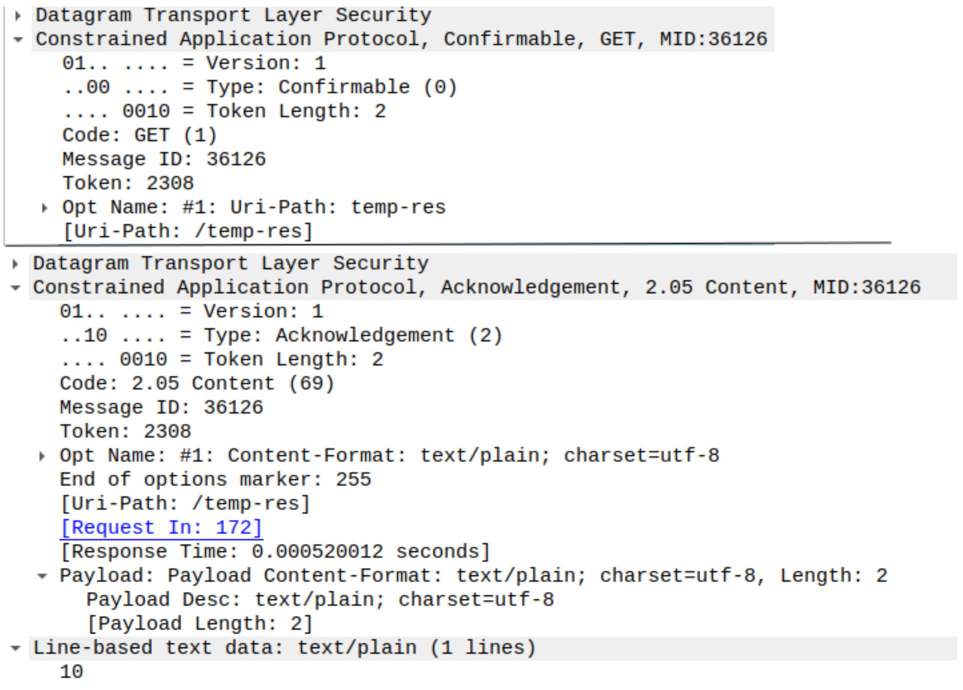
During development, the implementations of this work were continuously manually tested with each other for their proper functionality. These tests included different settings but were conducted without a reproducible and documented testing plan. Proper test cases would further ensure the robustness of the implementation, especially if they include error cases. Ideally, these tests could be run in an automated way. Additionally, these



```
▸ User Datagram Protocol, Src Port: 5683, Dst Port: 5683
▾ Constrained Application Protocol, Confirmable, POST, MID:36125
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0010 = Token Length: 2
    Code: POST (2)
    Message ID: 36125
    Token: 15f0
  ▸ Opt Name: #1: Uri-Path: authz-info
  ▸ Opt Name: #2: Content-Format: Unknown Type 61        Content-Format: application/cwt
    End of options marker: 255                           Payload: h‘d08344a1…5d82cc9b‘
    [Uri-Path: /authz-info]
    [Response In: 154]
  ▾ Payload: Payload Content-Format: Unknown Type 61, Length: 112
      Payload Desc: Unknown Type 61
      [Payload Length: 112]

0000  de df 0f e4 d9 fb a6 3e  f8 50 a2 2e 86 dd 60 00   ·······> ·P·.···`·
0010  00 00 00 8c 11 40 fe 80  00 00 00 00 00 00 a4 3e   ·····@·· ·······>
0020  f8 ff fe 50 a2 2e fe 80  00 00 00 00 00 00 dc df   ···P·.·· ········
0030  0f ff fe e4 d9 fb 16 33  16 33 00 8c 83 90 42 02   ·······3 ·3····B·
0040  8d 1d 15 f0 ba 61 75 74  68 7a 2d 69 6e 66 6f 11   ·····aut hz-info·
0050  3d ff d0 83 44 a1 01 18  18 a1 05 4c 00 00 00 00   =···D··· ···L····
0060  00 00 00 00 00 00 00 00  58 58 99 e3 bd 41 22 f0   ········ XX···A"·
0070  03 6c c6 5c 38 92 d7 74  2a 02 fb 79 88 0d 9a 30   ·l·\8··t *··y···0
0080  f6 93 24 4f 4d 9b 63 d3  d7 e0 1a 41 cf ef 0f e4   ··$OM·c· ···A····
0090  f2 9c 4a 1b 63 25 df 76  69 59 ff 31 f9 da 10 17   ··J·c%·v iY·1····
00a0  e2 70 be 9d 72 20 88 f3  97 66 fc d3 86 9d d2 f8   ·p··r ·· ·f······
00b0  e6 10 8e 00 9c c1 bb c4  ae d0 93 f1 0c 78 5d 82   ········ ·····x]·
00c0  cc 9b                                              ··

▸ User Datagram Protocol, Src Port: 5683, Dst Port: 5683
▸ Constrained Application Protocol, Acknowledgement, 2.01 Created, MID:36125
```

Figure 6.3: Upload of the Access Token that the client received with the response from the AS and the subsequent acknowledgment to the upload.

applications, which are implemented according to a standard, should also be able to interoperate with other implementations of the same standard. Such tests are to be conducted yet.



```
▸ Datagram Transport Layer Security
▾ Constrained Application Protocol, Confirmable, GET, MID:36126
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0010 = Token Length: 2
    Code: GET (1)
    Message ID: 36126
    Token: 2308
  ▸ Opt Name: #1: Uri-Path: temp-res
    [Uri-Path: /temp-res]
▸ Datagram Transport Layer Security
▾ Constrained Application Protocol, Acknowledgement, 2.05 Content, MID:36126
    01.. .... = Version: 1
    ..10 .... = Type: Acknowledgement (2)
    .... 0010 = Token Length: 2
    Code: 2.05 Content (69)
    Message ID: 36126
    Token: 2308
  ▸ Opt Name: #1: Content-Format: text/plain; charset=utf-8
    End of options marker: 255
    [Uri-Path: /temp-res]
    [Request In: 172]
    [Response Time: 0.000520012 seconds]
  ▾ Payload: Payload Content-Format: text/plain; charset=utf-8, Length: 2
      Payload Desc: text/plain; charset=utf-8
      [Payload Length: 2]
▾ Line-based text data: text/plain (1 lines)
    10
```

Figure 6.4: Requesting access to the resource within the *read* scope and the subsequent answer.

7 Evaluation

To evaluate the integration of the ACE framework into RIOT, the FIT IoT-LAB [21] testbed was used. The evaluations were run on IoT-LAB M3 boards. The settings used for the requests in this chapter are equal to the ones used in Section 6.2.

7.1 Protocol Flow Times

Figure 7.1 and 7.2 show the cumulative distribution of times that different kinds of requests take to complete. These time measures were started at the beginning of the client functions that prepare the requests and include everything until the end of the function that handles the incoming response from the server. As it can be observed, requests that include a DTLS handshake with asymmetric RPKs take up to 23 seconds until completion. Handshakes with a symmetric PSK take significantly less time. Requesting a token that is bound to a PSK takes on average 22.381 seconds when the client authenticates with a RPK. The same request, but authenticated with a PSK takes 89 milliseconds on average and therefore only about 0.4% of the time. A similar behavior can be observed for the resource access requests. With a RPK handshake this takes 22.526 seconds and with a PSK handshake only 73 milliseconds. For a resource access that is requested after the DTLS session was established, both methods take about the same time. This can be traced back to the procedure that during the RPK handshake symmetric session keys are calculated. Hence, the exchanged messages after the handshake are encrypted with symmetric keys, independent of the key type that was used for authentication.

Figure 7.3 shows how much time the program spends within the code that was implemented for this work. The difference to the completion times from Figure 7.1 and 7.2 consists of the time that is needed by the underlying protocols like DTLS and CoAP, as well as the transfer of the messages. This shows that the overhead introduced by the implementation is relatively small.

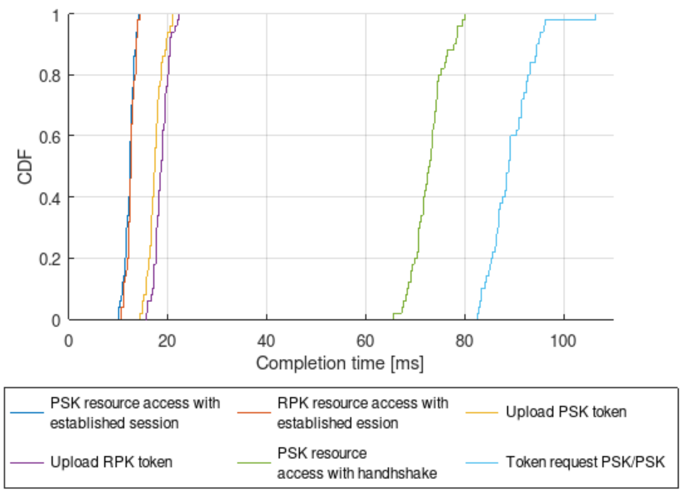


Figure 7.1: Completion times of full requests together with the respective evaluations and responses.

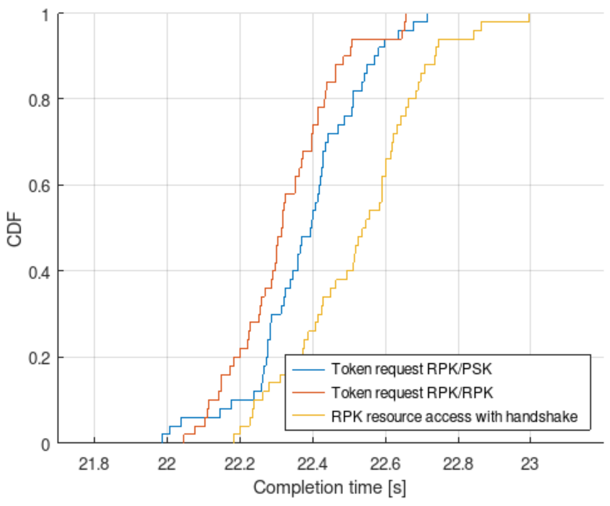


Figure 7.2: Completion times of full requests that need a bigger axis scale, i.e. requests that include a RPK DTLS handshake.

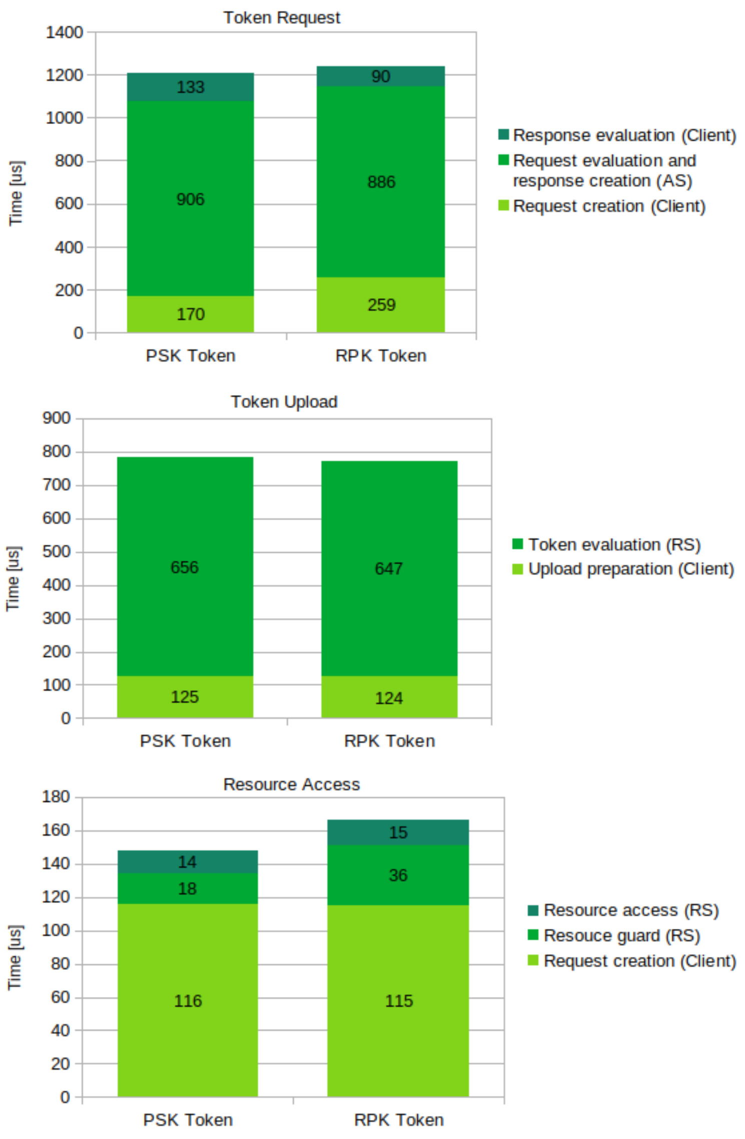


Figure 7.3: Average times that different parts of a token request, a token upload and a resource access take to complete. Only parts of the program that are implemented by this work are taken into account.

7.2 Firmware Size

Figure 7.4 shows the total required memory size of the firmware for the three different applications. In Figure 7.5 it is seen how much memory the RACED specific parts, as well as the necessary technologies for ACE need.

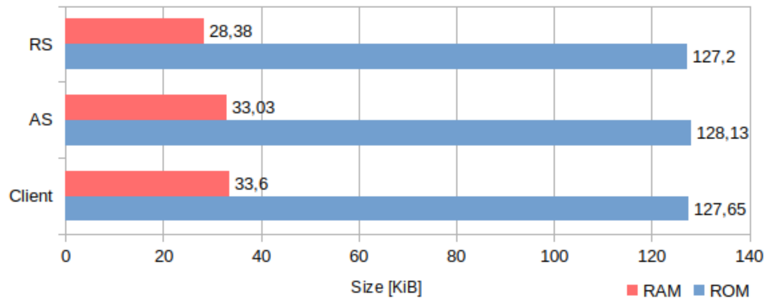


Figure 7.4: Total firmware size of the different entity applications.

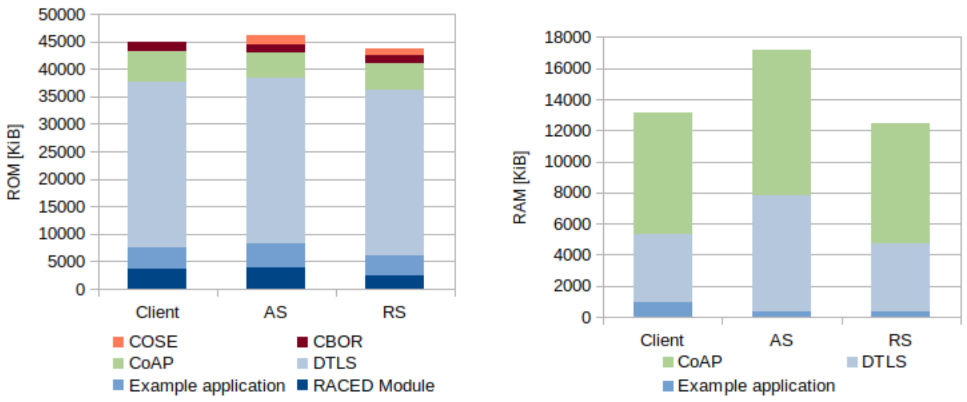


Figure 7.5: Required memory size for relevant parts.

8 Conclusion and Outlook

Developing an implementation of the ACE framework with RIOT profits from existing modules and packages of different standards that are used within the protocol flow of ACE. These can easily be incorporated into the implementation and alleviate the necessary work for a deployment. The RACED module further helps with the development as it fully abstracts the need to work with CBOR. Abstracting more parts of the framework into the RACED module turned out to be a difficult task since many aspects are highly dependent on the needs of the actual application. Therefore, this thesis implements applications for each of the ACE entities. These applications serve as examples that developers can use as guideline and build up on for their own deployment.

Even though the incorporation of modules is in general a major help for the development, it can also introduce restrictions in rare cases. For this work this especially comes up during the DTLS handshake. The inability to choose the key type within the application can become problematic for developers that want to program a real deployment of the ACE framework with RIOT (see Section 5.3). Further problems due to the inability to intervene with the handshake were shown in Section 5.4.

This work does not include some optional parts of the framework. Future work could therefore be conducted to implement the following concepts:

- Introspection
- Key and token expiration
- AS Request Creation Hint
- Key derivation
- Refresh tokens

Further, the support for other grant types, token types and profiles can be added. The RACED module offers a base for such extensions as it already implements CBOR encoding for all claims defined by RFC 9200 [1], 9201 [3] and 9202 [7].

As mentioned in Section 6.2, the functionality tests can also be expanded in future work. Another test scenario is the interoperability test with entirely different implementations of the ACE framework with the DTLS profile. One such implementation was developed by Marco Tiloca and Ludwig Seitz for Java. This implementation can be found at `https://bitbucket.org/marco-tiloca-sics/ace-java/src/master/`.

Bibliography

[1] L. Seitz, G. Selander, E. Wahlstroem, S. Erdtman, and H. Tschofenig, “Authentication and Authorization for Constrained Environments Using the OAuth 2.0 Framework (ACE-OAuth),” IETF, RFC 9200, August 2022.

[2] L. Seitz, S. Gerdes, G. Selander, M. Mani, and S. Kumar, “Use Cases for Authentication and Authorization in Constrained Environments,” IETF, RFC 7744, January 2016.

[3] L. Seitz, “Additional OAuth Parameters for Authentication and Authorization for Constrained Environments (ACE),” IETF, RFC 9201, August 2022.

[4] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach, Eight Edition, Global Edition.* Pearson Education Limited, 2022.

[5] J. Cowley, *Communications and Networking, An Introduction, Second Edition*, I. Mackie, Ed. Springer-Verlag London, 2012.

[6] R. Shirey, “Internet Security Glossary, Version 2,” IETF, RFC 4949, August 2007.

[7] S. Gerdes, O. Bergmann, C. Bormann, G. Selander, and L. Seitz, “Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE),” IETF, RFC 9202, August 2022.

[8] J. Postel, “User Datagram Protocol,” IETF, RFC 768, August 1980.

[9] E. Rescorla, H. Tschofenig, and N. Modadugu, “The Datagram Transport Layer Security (DTLS) Protocol Version 1.3,” IETF, RFC 9147, April 2022.

[10] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2,” IETF, RFC 5246, August 2008.

[11] Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP),” IETF, RFC 7252, June 2014.

[12] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)," IETF, RFC 8949, December 2020.

[13] J. Schaad, "CBOR Object Signing and Encryption (COSE): Structures and Process," IETF, RFC 9052, August 2022.

[14] M. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig, "CBOR Web Token (CWT)," IETF, RFC 8392, May 2018.

[15] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018. [Online]. Available: http://dx.doi.org/10.1109/JIOT.2018.2815038

[16] M. A. Ismail and T. C. Schmidt, "A DTLS Abstraction Layer for the Recursive Networking Architecture in RIOT," in *18. GI/ITG KuVS FachGespräch SensorNetze*. Universität Magdeburg, September 2019, pp. 25–28. [Online]. Available: https://doi.org/10.48550/arXiv.1906.12143

[17] D. Hardt, "The OAuth 2.0 Authorization Framework," IETF, RFC 6749, October 2012.

[18] M. Jones, L. Seitz, G. Selander, S. Erdtman, and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)," IETF, RFC 8747, March 2020.

[19] F. Palombini, L. Seitz, G. Selander, and M. Gunnarsson, "The Object Security for Constrained RESTful Environments (OSCORE) Profile of the Authentication and Authorization for Constrained Environments (ACE) Framework," IETF, RFC 9203, August 2022.

[20] O. Bergmann, J. Preuß Mattsson, and G. Selander, "Extension of the Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE) to Transport Layer Security (TLS)," IETF, RFC 9430, July 2023.

[21] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne, "FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed," in *IEEE World Forum on Internet of Things (IEEE WF-IoT)*, Milan, Italy, Dec. 2015. [Online]. Available: https://hal.inria.fr/hal-01213938

A Appendix

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original