CONNECT

CCAM TRUST & RESILIENCE

D5.2: Distributed Processing, Fast Offloading and MEC-enabled Orchestrator

Project number:	101069688
Project acronym:	**CONNECT**
Project title:	Continuous and Efficient Cooperative Trust Management for Resilient CCAM
Project Start Date:	1st September, 2022
Duration:	36 months
Programme:	HORIZON-CL5-2021-D6-01-04
Deliverable Type:	OTHER
Reference Number:	D6-01-04 / D5.2 / V1.1
Workpackage:	WP 5
Due Date:	29th February, 2024
Actual Submission Date:	11th March, 2024
Responsible Organisation:	ICCS
Editor:	Panagiotis Pantazopoulos
Dissemination Level:	PU - Public
Revision:	V1.1
Abstract:	D5.2 follows up the D5.1 contributions to deliver the first version of the CONNECT orchestrator and the mechanisms for secure data exchange. The here-in reported work covers the task-offloading logic and the implementation details of the CONNECT orchestartor software. Lastly, the work is completed with the Blockchain architecture and data handling principles to facilitate the CONNECT accountable attestation and monitoring (when combined with the D5.1 claims).
Keywords:	Task offloading logic, CONNECT (confidential) containers orchestration, Blockchain design, smart contracts and data management

Funded by the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or CINEA. Neither the European Union nor the granting authority can be held responsible for them.

Editor

Panagiotis Pantazopoulos (ICCS)

Contributors (ordered according to beneficiary numbers)

Stefanos Vasileiadis, Anna Angelogianni, Nikos Fotos, Thanassis Giannetsos (UBITECH)
Panagiotis Pantazopoulos, Pavlos Basaras, Vangelis Kosmatos (ICCS)
Konstantinos Latanis, Giannis Constantinou (Suite5)

Disclaimer

The information in this document is provided as is, and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author's view – the European Commission is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability.

Executive Summary

The document constitutes the second deliverable of the CONNECT WP5 work gathering the project achievements along three main research fronts:

The design of the task offloading logic -as a generic theoretical algorithm- to identify the needs and steps of shifting demanding tasks to the infrastructure. Then, relying on the proposed algorithm, the identification of practical engineering challenges to be addressed in the subsequent WP6 real-world (SMTD) demonstrator.

The detailed software design and specification of involved tools to realise the MEC-enabled CONNECT Orchestrator which deploys software to all automotive points of interest (i.e., vehicle platform, edge and cloud infrastructure). Detailed descriptions highlight the readiness level of the first version of the CONNECT Orchestrator.

Finally, the CONNECT blockchain solution is motivated and specified (while its implementation will be detailed in the final WP5 deliverable). Its architecture and the way to fulfil the CONNECT needs is explained. Furthermore, the involved data requirements are defined together with the required smart contracts. The work concludes with CONNECT blockchain data management choices in terms of access control policies and encryption attributes.

The work presented in the deliverable at-hand, together with the D5.1 inputs allows the project a) to realise the (final) CONNECT confidential containers orchestration layer, and b) build the CONNECT blockchain overlay to store CONNECT claims and ensure auditability. (Note that the security mechanisms included in the CONNECT TEE guard and employed to construct the D5.1 verifiable credentials and claims are detailed in D4.2) As such, the project is enabled to move towards the final WP5 technology outcomes to feed the forthcoming CONNECT integration and demonstration.

Contents

1 Introduction **1**

1.1 Scope and Purpose of the Document 2

1.2 Relationship with other CONNECT Deliverables 2

2 The CONNECT Task-Offloading **4**

2.1 Motivation and needs 4

2.2 The CONNECT task-offloading: a generic algorithm 5

2.2.1 Focus points, modeling assumptions and remarks 5

2.2.2 Proposed offloading algorithm 7

2.3 The CONNECT task-offloading in practice: compliance with available data and resources 8

3 The CONNECT Containers Orchestrator **10**

3.1 CONNECT Orchestrator software design and development 10

3.1.1 The CONNECT approach to container orchestration 10

3.1.2 K3s Overview 11

3.1.3 K3s Architecture 12

3.2 Securing Application Containers Technology in the CONNECT set-up 14

3.2.1 Building the CONNECT (K3s) Confidential Containers 14

3.2.2 K3s in CONNECT: Cloud, MEC and vehicle nodes 15

3.2.3 CONNECT Confidential Containers Flow 19

3.3 CONNECT Orchestrator in Lab Environment 21

3.4 Towards the CONNECT task-offloading realisation 24

4 The CONNECT Blockchain Infrastructure for Auditability **26**

4.1 The Blockchain motivation in CONNECT 26

4.2 The Blockchain architecture and the CONNECT approach 27

4.2.1 Overview on Blockchain technologies for CONNECT needs 27

4.2.2 Architecture 28

4.3 Blockchain data requirements 30

4.4 Definition of Smart Contracts 31
4.4.1 Smart Contract deployment workflows 31
4.4.2 Data storage in CONNECT DLT (attestation evidence) 34
4.4.3 Data retrieval in CONNECT DLT 35
4.4.4 Smart Contract Data Model Definition 36

5 Blockchain Data Management **40**
5.1 Attribute-based Access Control 40
5.2 Attribute-based Encryption 42
5.2.1 Ciphertext-Policy ABE 43
5.2.2 Key-Policy ABE 43
5.2.3 Decentralized ABE in CONNECT 43
5.2.4 Overview of the CONNECT ABE scheme 45

6 Conclusions **51**

7 List of Abbreviations **52**

Bibliography **56**

List of Figures

1.1 Relation of D5.2 to other deliverables and WPs 2

2.1 An offloading set-up of multiple MEC servers and involved delays 6
2.2 The process required for having the $T3$ available at the vehicle 6
2.3 Task-offloading may include a second communication leg and some in-vehicle computations (grayed-out part) 8

3.1 K3s Architecture 12
3.2 K3s Architecture – Single Server, Multiple Agents 13
3.3 CONNECT orchestration solution 15
3.4 K3s generation of self-signed CA 16
3.5 Service instantiation in confidential containers 20
3.6 K3s worker node and Private 5G network at ICCS premises. 22
3.7 K3s master node hosted at ICCS servers. 23
3.8 K3s master node services for LCM support of cluster operations. 24

4.1 CONNECT DLT Architecture 28
4.2 Smart contract transformation 32
4.3 Smart contract execution 33
4.4 Attestation evidence recording in CONNECT DLT 35
4.5 Retrieval of data from CONNECT DLT 36

5.1 ABAC mechanism in CONNECT DLT 42
5.2 Overview of the CONNECT ABE scheme 46
5.3 Access control tree created by the SCB 47

List of Tables

4.1 CONNECT DLT Requirements 31
4.2 Confidentiality and Integrity Requirements 32
4.3 Authorisation and Access Control requirements 33
4.4 Encryption requirements 33
4.5 Non-repudiation and Accountability requirements 34
4.6 Trust Policy Model Structure 37
4.7 Attestation Report Data Structure 39
4.8 AttestationPublicKeys Data Structure 39

5.1 Notations used in the description of the CONNECT ABE scheme. 46

Chapter 1

Introduction

The document at-hand constitutes the second outcome of CONNECT WP5. It serves as a second yet fundamental WP5 pillar towards the final outcome(s) of this workpackage.

In terms of its content, the deliverable gathers two main threads of the WP work: the first one (Sections 2 and 3) refers to the task offloading process relying on the D5.1 state-of-the-art analysis. A generic algorithm for task-offloading is followed by practical considerations accounting for the characteristics of the CONNECT set-up and highlighting the dimensions to be evaluated. Moreover, the implementation details of the CONNECT containers' orchestrator are discussed. The employed orchestration technology is motivated and described while particular emphasis is placed on its relevance to the concept of confidential containers and the way their deployment is engineered. A brief presentation of the current in-lab CONNECT deployment highlights our progress towards the final D5.3 release. The aforementioned work corresponds to the ongoing tasks 5.1-5.3.

The second part (Sections 4 and 5) motivates and discusses the CONNECT Blockchain technology. It details the progress along the CONNECT Blockchain infrastructure where accountable trust assessment information and attestation-related evidence are to be stored and shared. The requirements (introduced in D2.1) are refined while smart contracts and the way to manage the involved data are detailed. That work relates to T5.3 but mainly corresponds to T5.4 achievements.

Looking at the big picture, the work presented here allows the project to proceed to its final implementation effort towards :

- the MEC-enabled virtualized resources orchestrator to drive the CONNECT functionality (along all automotive points of interest i.e., vehicle, MEC and cloud) and support its task offloading capability in the considered automotive setting.
- the design and development of Blockchain-based technology to monitor/store CONNECT trust-related and attestation data, ensuring accountability of the automotive actors.

Along these lines, D5.2 sets the basis for the finalization of the CONNECT (MEC-enabled) communication technology; the orchestration of confidential containers and the (different dimensions of the) secure data sharing concept. The latter involves the verifiable presentations (see D5.1) and the capability for auditable storage of claims through the Blockchain usage. Those contributions will help WP5 deliver its final outcome (D5.3 release) and most importantly, offer the whole communications stack to the WP6 demonstrator.

1.1 Scope and Purpose of the Document

The document first aims to describe the CONNECT task-offloading; a general CCAM solution as well as the practical considerations tailored for the CONNECT demonstration set-up, are detailed. It also seeks to cover the application containers ICT technology explaining the CONNECT mechanisms required for confidential deployments. Finally, the deliverable motivates the usage and describes the functionality of a Blockchain layer to keep the CONNECT attestation results and trust-related data.

Overall, the work carried out so-far and described in this deliverable, allows CONNECT to a) move towards the final version of (MEC-enabled) orchestrator for managing confidential containers and b) introduce all design principles and Blockchain data management algorithms/controls for auditable CONNECT data exchanges.

1.2 Relationship with other CONNECT Deliverables

The deliverable presents the work that acts as the solid basis for the finalisation of the WP5 implementation work (to be reported in D5.3).

On the "horizontal axis" (see Figure 1.1), it draws on the outcome of the D5.1; the so-far background of offloading solutions helps us suggest a task-offloading logic appropriate for the automotive setting. Moreover, D5.1 introduces the verifiable credentials and presentations which offer a way for trust-related data to be captured and securely pushed to the CONNECT Blockchain (see Chapter 5). Accordingly, D5.2 presents the first version of the CONNECT orchestrator and the Blockchain network. These will act as a basis for their final version to be detailed in D5.3.

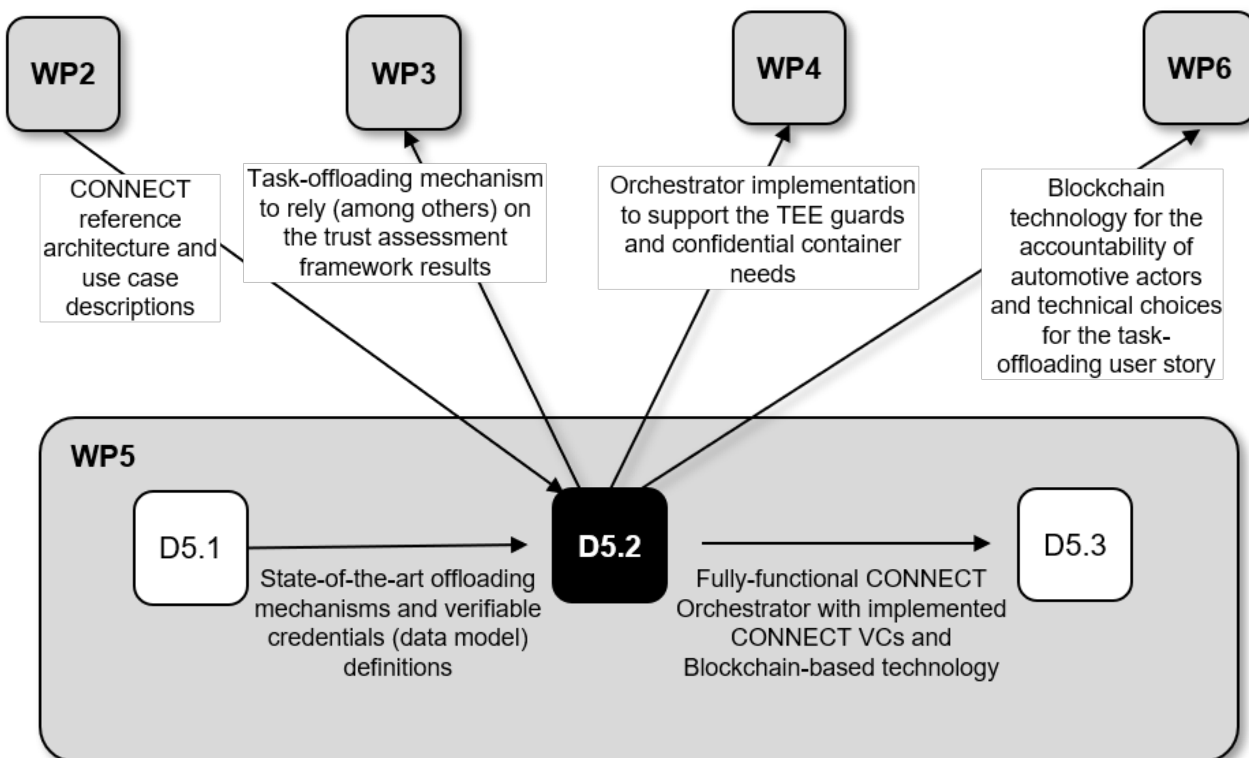

Figure 1.1: Relation of D5.2 to other deliverables and WPs

On the vertical axis (of Figure 1.1), the deliverable builds on top of the WP2 specifications work (i.e., MEC requirements, architecture and use-cases introduction, detailed in D2.1). Subsequently, D5.2 provides insights to the WP3 TAF development and deployment, which should be closely coupled with the operation and needs of CONNECT confidential containers. In parallel, the implemented modules of the CONNECT orchestrator will follow the confidential communication principles and rely on the corresponding technology (e.g., enclave-cc [27]), as determined in WP4 (D4.2). Finally, the integration tasks (D6.1) and use cases analysis (D6.2) will be based on the final (i.e., integration-ready) version of the CONNECT application containers orchestrator as well as the blockchain layer, introduced in this deliverable.

Chapter 2

The CONNECT Task-Offloading

2.1 Motivation and needs

The concept of task-offloading (see D5.1 definition) holds a significant role in the CONNECT vision. It involves the shift of tasks from the vehicle(s) to the infrastructure where the computational resources are typically richer. Subsequently, the result of the task execution may need to be returned to the original vehicle (and/or neighboring vehicles), or, trigger other actions (such as the communication to a cloud CCAM server).

The main motivation to employ task-offloading operations is (in line with the D5.1 discussed background) the increased in-vehicle computation needs and the ever-increasing complexity/computational requirements of the CCAM applications, which may also combine information accumulated from other vehicles to deliver a consolidated service/view for connected vehicle applications. In view of the higher automation level of connected vehicles (mainly with the inclusion of advanced ML-based, cooperative perception and positioning capabilities [21]), further computational needs are posed, even overcoming the proliferation (rate) of capable in-vehicle devices/hardware [19]. Other needs such as the requirement for a certified[1] vehicle operation can also motivate task offloading. Finally, reasons such a faulty vehicle device or the identification of compromise serve as triggers for task migration rather than offloading (as defined in D5.1)

Interestingly, the actual task to be offloaded may not only involve a CCAM service or the associated CCAM data; the actual trust computations of the CONNECT framework (i.e., information stemming from the TAF data sources, as explained in D3.2) may also be the subject of the task-offloading function. Centrally gathering and fusing trust opinions may become beneficial for dynamic trust assessment purposes (see D3.2). In all of the above cases, it is straightforward to expect that it is the latency of the offloading process which holds the critical key-role[2] in the given environment (see relevant comments in Section 2.3). It follows that a potentially increased time needed (i.e., delay) caused by network latency or heavy trust-related computations, may render the offloading functionality impractical.

The above discussion points to the following main needs for the task-offloading operation:

[1] UN R155 [1] suggests the existence of a cybersecurity management system applied through the whole life cycle of a vehicle. Then, (the cybersecurity) operation of each vehicle platform is certified using ISO/SAE 21434 [2] means. In case a computation task (of a CCAM application) has not been considered in the (latest) certification process, offloading it to the infrastructure could be a way for the vehicle to remain compliant with its latest certificate.

[2] Energy consumption is considered in literature [20] as almost equally-important for task-offloading. However, it lies outside the CONNECT scope.

- Data availability on multiple involved quantities such as CPU utilisation, memory, network resources, bandwidth etc (see D5.1 and the rest of this chapter). All of those constitute input-data to the task offloading process and are typically collected and become available through telemetry means [32].
- A decision algorithm that based on the accumulated telemetry data and the service requirements (e.g., end-to-end service delay) decides either to offload a task to the infrastructure, or execute it locally.
- The task-offloading functionality would in general require the relevant software instances residing both at the vehicle platform and the infrastructure. Accordingly, the relevant (HW/SW) modules would need to be assessed in terms of trustworthiness; that points to the notion of the federated trust assessment (TAF), as discussed in D3.2.

In more practical terms, these dimensions and the extent to which each of them is going to be fulfilled, is expected to be shaped by the characteristics and limitations of the studied use-case.

2.2 The CONNECT task-offloading: a generic algorithm

This section adopts a generic (automotive) standpoint whereby a vehicle and multiple candidate offloading locations at the (infrastructure) may be available [3]. While this approach would require the availability of multiple MEC locations and apparently the support of an MNO, we choose to elaborate on it in order to gain helpful insights towards more practical considerations (in Section 2.3).

2.2.1 Focus points, modeling assumptions and remarks

The main question in the considered setting of the in total K candidate MEC locations (see Figure 2.1) is a way to identify the most prominent one to offload a given task; prominent may have numerous meanings in line with the objective (see D5.1) that is -each time- to be minimized. In our case, as earlier discussed, time (in terms of delay) is the central parameter that CONNECT aims to study and ”correlate” with the involved trust computations.

[3]The considered setting may become even more complex under the assumption that multiple MNOs as well as multiple MEC service providers are offering the cellular network connectivity and the (offloading) destination platform, respectively. To address such an approach would call for resorting to impractical optimizations [17] as well as insights provided by an MNO and/or MEC service provider; both options go beyond the CONNECT scope.

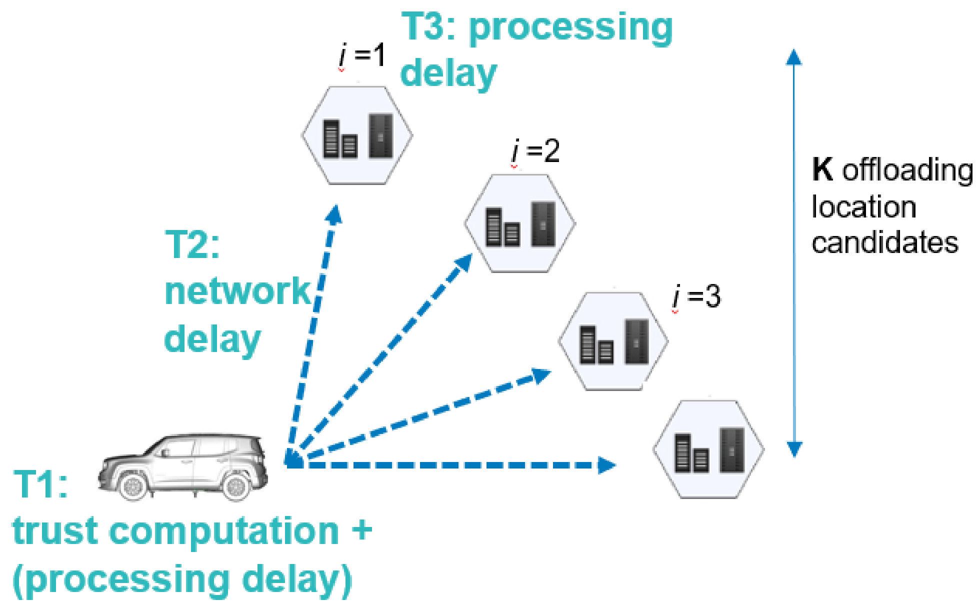

Figure 2.1: An offloading set-up of multiple MEC servers and involved delays

We therefore, identify (in Figure 2.1) three main time periods (delays) that are relevant to the task offloading operation. The time $T1$ required to carry-out any local (i.e., in-vehicle) processing and trust-computation, the time $T2$ which is the network delay (for application and trust data) to reach the MEC server and finally, $T3$ which reflects the time needed for the offloaded task to be carried out by the MEC-hosted software. Under this setting, the main research question is the identification of the i-th MEC server which enables the minimization of the sum of all times $T1 + T2 + T3$.

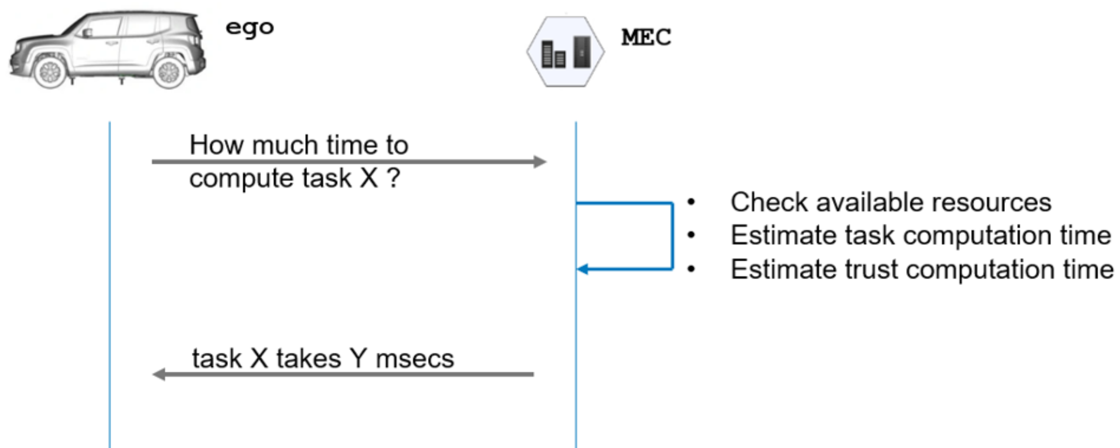

Figure 2.2: The process required for having the $T3$ available at the vehicle

Our analysis adopts as a starting point the fact that the task-offloading decision (i.e., whether to perform task offloading or execute locally on the vehicle) takes place at the vehicle, where all the relevant data should be gathered. This approach is based on the following points:

1. safety reasons [33] prevent the remote management of tasks, which is inline with OEM strategies suggesting that critical decisions remain with the vehicle [22]

2. any (critical) decision if taken at the infrastructure side would require a further increase of the uncertainty (i.e., typically present in any communication link) and potential risks.

3. the input-data (e.g., sensor readings) for any driving decisions are directly available in the vehicle while requiring time to be sent to the infrastructure; even if this time is minimal, significant changes may appear in the environment [3] (due to increased vehicle speed).

4. in line with the previous reasons, the majority of the relevant background (see D5.1) considers the vehicle as the decision-making point for task-offloading.

The same approach will be adopted in our practical task-offloading consideration (see section 2.3) as well as the CONNECT task-offloading demonstrator (in WP6).

2.2.2 Proposed offloading algorithm

The proposed algorithm (presented in as pseudo-code in Algorithm 1) is a simple greedy heuristic algorithm which relies on the idea that $T1$ and $T2$ can be estimated by the vehicle, while $T3$ may essentially attain any value in $(0, +\infty)$; this is because it remains subject to the MEC resources that vary (rather unpredictable) in the course of time. To obtain a $T3$ value, a process like the one depicted in Figure 2.2 needs to be followed for every candidate server; that identification task would apparently add further delays.

On the contrary, the sum of $T1 + T2$ can be known to the vehicle by a) any type of benchmarking [12] of the local trust (TAF) computations and b) by obtaining round-trip delay estimations with the usage of tools such as the `ping` utility or any correlation with the distance-to-base, typically provided by off-the-shelf 5G modems [30]. Along these lines, the sum $T1 + T2$, called $PartialSum_i$ for server i (see algorithm 1, line 2), can act as the main criterion for the identification of the most appropriate MEC server, greedily estimating the process time $T3$.

Algorithm 1 *Task-offloading selection* $(set\ of\ K\ servers)$

1: Estimate $sum_i(T1_i, T2_i)$ for $i = 1$ to K
2: Vector $PartialSum_i \leftarrow sum_i(T1_i, T2_i)\ \ \forall i$
3: Rank vector $PartialSum_i(.)$ from lowest to higher values
4: **for** j=1 to K **do**
5:     ping server i corresponding to $PartialSum_i(j)$ and estimate $T3(i)$
6:     **if** $PartialSum_i + T3(i) \leq THRS$ **then**
7:         Terminate the process and select server i
8:     **end if**
9: **end for**

Therefore, having identified a vector of $T1 + T2$ values (line 2) and subsequently ranked that vector with respect to the $PartialSum_i$ =$(T1_i + T2_i)$ for every candidate server, the proposed algorithm suggests a sequential query (line 4-5) to every candidate MEC location, starting from the one with the lowest $PartialSum$. The query involves on its time needs (i.e., $T3$ as depicted in Figure 2.2) to execute the task (to be offloaded). Then, a greedy stopping condition (line 7) for the search process suggests to conclude the process by the time that server i which provides a $T3_i$ value such as the total sum of delays i.e., $PartialSum_i + T3_i$ is lower than $THRS$, has been identified. The $THRS$ threshold value marks the tolerable delay by an automotive application to

have a task offloaded and carried-out at the infrastructure. Clearly, there are no guarantees that the identified server is the one yielding the globally minimum total delay.

The corresponding threshold $THRS$ clearly depends on the involved in the offloading (automated) driving application and its safety requirements (see [7] or [3]). It is safe to assume that the relevant $THRS$ is a-priory known for all the current (even day-2 [31]) driving applications. Finally, as regards the proposed algorithm's time complexity, it is straightforward that the worst case scenario involves the estimation of cumulative $T1 + T2$ delays for the K servers and then the query for $T3$, once again, of all K servers. That is $O(2K)$=$O(K)$, resulting in a welcome linear algorithm with respect to the number of different MEC location candidates.

2.3 The CONNECT task-offloading in practice: compliance with available data and resources

We now proceed to more practical considerations of the task offloading process in order act as a basis for our real-world experimentation and demonstration (WP6). Along this line, we first point to some important insights provided by our analysis of Section 2.2.

In real practice, the result of the execution of an offloaded (to the infrastructure) task may trigger a communication (i.e., sending the task results) back to the vehicle(s), to road users, or upwards to cloud-based services. The former case is depicted in Figure 2.3 where the delays (i.e., $T4$ and $T5$) of the second leg (i.e., from the edge server to the vehicle) are shown. Those however, are quantities that are either application-specific; meaning that the time $T5$ for the vehicle reaction is subject to the currently enabled automotive application in the vehicle. Or, the vehicle position affecting the $T5$ value, is not accurately known. Clearly, for both $T4$ and $T5$ it is non practical/(feasible) to be measured. As such, in the experimental analysis of the offloading we will focus on the three ($T1, T2$ and $T3$) delays and neglect the ones appearing in the grayed-out part of Figure 2.3.

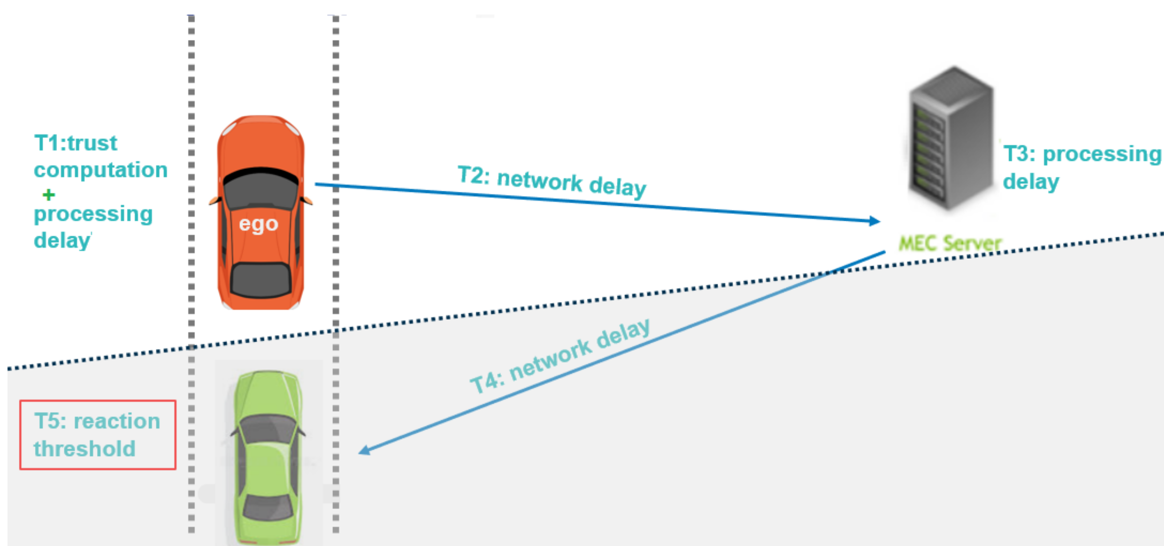

Figure 2.3: Task-offloading may include a second communication leg and some in-vehicle computations (grayed-out part)

Considering the (Section 2.2.2) automotive setting shown in Figure 2.1, we highlight some points of significance for the CONNECT real-world (experimentation) environment:

- The available resources (and consortium line-up) suggest that the experiments would be carried-out having a 'single cell' set-up. That is posed by the available experimentation hardware set-up but also the lack of MEC service provider(s) or MNOs in the CONNECT partner profiles. That means that our focus will be placed on offloading engineering aspects under the case of a single offloading server. That option may prevent the experimentation with multiple MNO trust domains or the "follow-me-Edge" functionality [4] but allows the exploration of open fundamental questions (as abstracted in the D2.1 SMTD user-story 5).
- Regardless the actual destination of the offloaded task result i.e., whether sending it back to the vehicle(s) (for instance as shown in the lower part of Figure 2.3) or forwarding the result to a cloud server, the decision to offload a (CCAM) task or not, will be mainly shaped by time requirements (when MEC resources are sufficient). In all cases, when trying to minise the total time for having a task computed at the MEC, the tolerance/need for longer yet more demanding trust computations (included within the $T1$ and potentially $T3$ delays) needs to be assessed in relation to the achievable network delay ($T2$) to have the task shifted at the MEC. The CONNECT experimentation aims to shed light on those (in-practice) under-explored directions.
- The estimation of the in-vehicle computations (i.e., taking $T1$ time in Figure 2.3) call for benchmarking of trust computations (see Section 2.2.2). Those would involve the vehicle platform but could also extend to the MEC server (to be included in $T3$), if the notion of federated TAF (D3.2) becomes relevant.
- Clearly the practical requirement of having all input-to-the-offloading-decision data available, is to be fulfilled by monitoring and telemetry tools (see Section 3.4). Such data include the computational resources (e.g., CPU utilisation, memory, etc) at the vehicle platform and the infrastructure side which are typically collected by native K3s tools and are designed for seamless integration and straightforward data gathering. However, alongside this, we recognized the need for a more complex telemetry system, which will be discussed in D5.3, for collecting network-related telemetry data, for example. Although most theoretical and simulation studies (see D2.1) consider these data readily available, in practice, it remains challenging to collect them.

All above points essentially constitute open research questions for WP6 work. Importantly, most of the above directions (e.g., the achievable end-to-end latency) have been already identified in the D2.1 KPIs.

On a final research-related note, the task-offloading process has been extensively studied in literature, mainly through modeling tools and simulation (see D5.1). However, there is a very limited set of works providing results from real-world experiments (e.g., [34] or based on real-world data [5]). Our WP6 work will contribute towards addressing this research gap.

Chapter 3

The CONNECT Containers Orchestrator

This chapter details the progress on the CONNECT orchestration technology. It highlights the tools employed to create and manage confidential containers. Furthermore, it elaborates on the relevant deployment options in each of the CONNECT points of interest (i.e., cloud, edge and vehicle) discussing an indicative example of the involved flow of actions for the orchestration and management of confidential containers, as well as the relevance to the task-offloading demonstrator.

3.1 CONNECT Orchestrator software design and development

In what follows, the CONNECT orchestrator instance is motivated and detailed from an architectural point of view.

3.1.1 The CONNECT approach to container orchestration

The CONNECT orchestration solution will not be built from scratch, but will be realised on top of already available (state-of-the-art, SoTA) container management and orchestration platforms. In the project, the cloud-native approach is adopted, in which all modules and functionalities presented in detail in D2.1 will be developed as lightweight services (”Microservices”) and deployed as software containers. These software containers will be lightweight services, loosely coupled, as per the cloud-native paradigm, able to communicate with in-cluster secure (CONNECT) container services and legacy applications, but also, if necessary, with the outside world.

In general, Kubernetes (K8s) is considered as the SoTA container orchestration platform, providing automated features for deployment and management, scaling, networking, and load balancing functionalities for the containers, the applications as well as the infrastructure nodes that host them. The “regular” K8s container orchestration platform carries a heavy CPU and memory footprint as it is designed for large-scale cloud computing clusters. Therefore, in the CONNECT case, where a plethora of resource-constrained edge devices exist (e.g., the on-board units - OBUs on CONNECT vehicles), we explored a more lightweight solution.

Particularly, in the last few years several lightweight Kubernetes distributions have emerged [8]. Indicatively such distributions include Minikube, MicroK8s, K3s, k0s, KubeEdge, or MicroShift, to mention just a few of them. These solutions are targeting to support container orchestration

also on resource-constrained devices at the edge (e.g., at the MEC) or at the far-edge, i.e., at the focal point closer to the end-sensors where data are generated (e.g., on vehicle). This approach is deemed as an appropriate fit for the CONNECT case and CCAM applications.

Especially for resource constrained edge devices in factories, autonomous cars, or smart cities, the performance overhead added because of the container orchestration functionalities may not be negligible [15], therefore the selection of the appropriate orchestration platform is important.

To help with this selection a short analysis was realised. The analysis took into consideration the CONNECT orchestration requirements and based on the following aspects concludes the lightweigth Kubernetes distribution to be used in CONNECT: a) papers presenting comparison between lightweight Kubernetes distributions; b) compatibility issues and device dependencies; c) past experience is using selected distributions.

Regarding the first aspect, papers [8], [15] and [16] present extensive comparisons between different lightweight Kubernetes distributions. In [8] a comparison between MicroK8s and K3s is executed. The paper concluded that K3s has better performance. In [15] the authors evaluate the performance of three different Kubernetes distributions: full-fledged Kubernetes, K3s, and MicroK8s. The results demonstrate that both K3s and MicroK8s have not only managed to reduce the deployment time and complexity but have also improved performance. In [16] a comparison of MicroK8s, K3s, k0s, and MicroShift distributions is realised, by investigating their minimal resource usage as well as control plane and data plane performance in stress scenarios. In stress scenarios, K3s and k0s marginally showed the highest control plane throughput compared to the others.

Regarding the compatibility aspect, the Vehicle OBU utilised in the CONNECT scenarios can support the K3s distribution, therefore this distribution is preferable against the others. More information on these aspects will be presented in details in deliverable D6.1.

Regarding the third aspect, ICCS (WP leader) is leading the development of the kubernetes cluster that will support CONNECT and has extensive experience with the majority of these distributions. Hence, we conclude on the utilization of K3s that is a certified Kubernetes distribution designed for production workloads in unattended, resource-constrained, remote locations, as the *CONNECT orchestration platform*.

3.1.2 K3s Overview

In detail, K3s was released by Rancher in 2019 and then acquired by SuSE in 2020. The team behind K3s wanted an installation of Kubernetes that was half the size in terms of memory footprint. In this direction, Rancher has packaged K3s as a single binary (approx. 65 MB) with basic K8s components. Thus, it does not require a package manager for installation and is independent of any particular Linux distribution. It is fully compliant to K8s, contains all basic components by default, and targets a fast, simple, and efficient way to provide a highly available and fault tolerant cluster to a set of nodes. K3s replaced etcd by another datastore, the sqlite3 which is far more lightweight. K3s tries to lower the memory footprint by a reorganization of the control plane components in the cluster. The K3s master and workers, also called server and agents, encapsulate all the components in one single process. It is written in Go and has a reported memory footprint of 512 MB RAM, with the developers recommending at least having 1 GB of RAM for regular deployments and 4 GB of RAM for high availability installations [16]. Based on the paper in [16] which compares different lightweight Kubernetes distribution, it is by far the most popular lightweight K8s distribution with more than 20,000 stars and over 1700 contributors on Github.

3.1.3 K3s Architecture

The high-level architecture of K3s is illustrated in Figure 3.1. All the three figures of this subsection are coming from the original on-line K3s documentation and present the architecture and most common configurations of K3s distribution. The main building blocks of the K3s architecture are the Server and the Agent, similarly to the building blocks of Kubernetes which are the Master node and Worker node. In the K3s deployment illustrated in Figure 3.1, we assume one K3s Server and one K3s Agent. In this section an overview of the K3s distribution is presented. The detailed description of the modules illustrated in Figure 3.1 is presented in section 3.2.2.

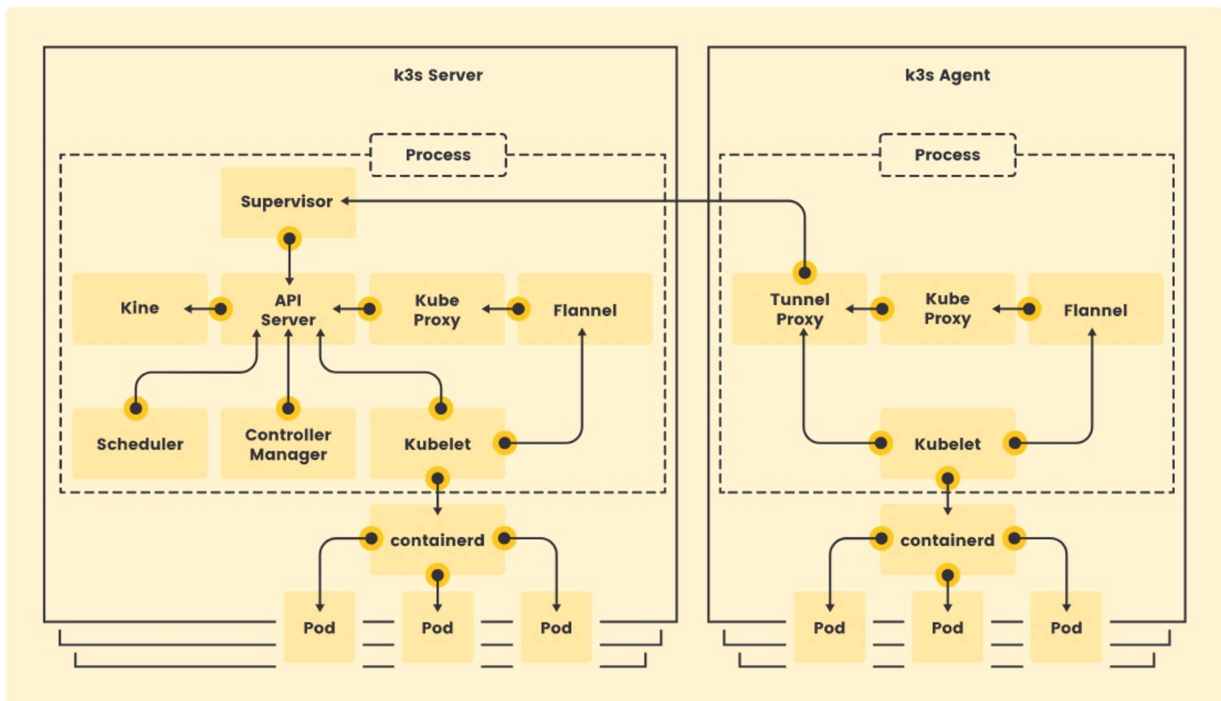

Figure 3.1: K3s Architecture

A K3s Server and a K3s Agent communicate through a proxy called Tunnel Proxy as depicted in Figure 3.1. The first interaction is initiated from the K3s Agent (i.e., so as to add the agent/work node to the cluster) which communicates with the K3s Server and sets up the tunnel. All data exchanged between Server and Agent are realized through this Tunnel Proxy, which is responsible for establishing the connectivity between the appropriate ports that facilitate the relevant control plane functionalities of the cluster.

In the Kubernetes case, the Kube Proxy uses several ports to get connected with API-Server, but in the K3s case the Kube Proxy gets connected with API-Server using the Tunnel Proxy. In detail, the Tunnel Proxy creates a unidirectional connection toward the API-Server. Then since this link is established, a bidirectional communication is realised. In this way, a single port is used for connectivity between the elements, making the connectivity more secure.

In Kubernetes the etcd component is used, which is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. In the K3s case the etcd is replaced by an SQLite database. Even better, the Kine can be used in K3s, which is a datastore that allows etcd to be replaced with other databases (e.g. MySQL, PostgreSQL, SQLite, and more). It acts

as a translation layer that allows K3s to use these databases, instead of etcd that is used in full kubernetes.

Similar to the Kubernetes case, K3s uses Kubelet. In practice Kubelet is an agent that runs on each node in the cluster. Kubelet makes sure that containers are running in a Pod (which in simple terms, is an abstraction layer[1] introduced in kubernetes, to be agnostic to the container technology used for building the container image, e.g., docker, linux container or other). In addition, the kubelet takes a set of Pod specifications that are provided through various mechanisms and ensures that the containers described in those specifications are running and healthy.

In K3s, as in a typical K8s cluster, network plugins are used (e.g., Flannel), which act as a container network interface (CNI) for cluster networking inside the K3s environment. Flannel interacts with Kubelet and also with Kube Proxy. In both Kubernetes and K3s, the container runtime is tasked with the life cycle management operations of the host nodes and their workload, i.e., their containers. Kubernetes support various implementations for the container runtime module including Containerd and CRI-O. In the Kubernetes case, all the components are running in different processes, while in K3s case, all the aforementioned components run together as a single process which makes it really lightweight.

In Figure 3.2 a K3s cluster case is illustrated in which a single K3s Server is connected to a set of K3s Agents. As illustrated in the figure, the K3s embedded SQLite database is used for storing all the information related with the K3s operations. In this configuration, each agent node is registered to the same server node. The management of the K3s resources can be realized by a user through the K3s API exposed on the server node.

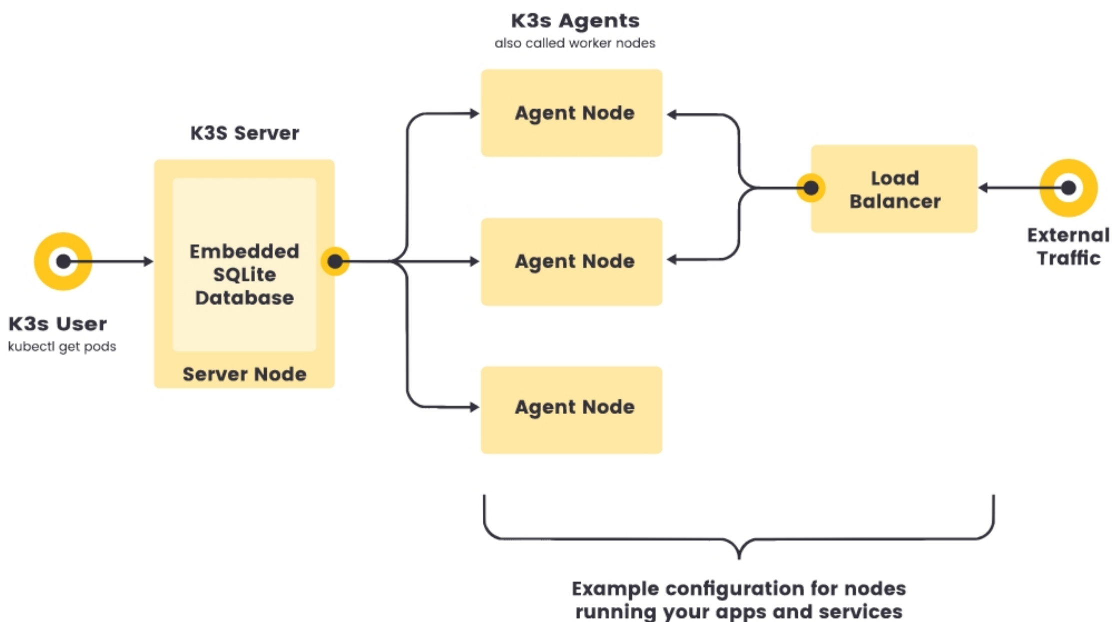

Figure 3.2: K3s Architecture – Single Server, Multiple Agents

For the needs of the CONNECT project and the defined Use Cases, in which the deployment of services is foreseen for three different locations: cloud, MEC, vehicle, the aforementioned K3s deployment and configuration is adequate to fulfill the requirements. Therefore, the above

[1] https://kubernetes.io/docs/concepts/workloads/pods/

configuration will be used for all the related Use Cases. More information will be provided in the next sections.

3.2 Securing Application Containers Technology in the CONNECT set-up

Prior to the description of the building phase of the confidential containers that follows in Section 3.2.1, we provide some further details on the LCM operations in kubernetes that are facilitated through the container runtime (and its breakdown in various smaller modules, namely runc, shim and containerd), as they also play a crucial role in the building phase of the CONNECT containers. Particularly, runc is a powerfull tool for spawning and running containers according to the OCI (Open Container Interface). It is the default runtime for Docker and containerd, and is responsible for the actual running of the containers. In this context, shim is a small intermediary program that sits between the container runtime (e.g., containerd) and the actual container process. Its main purpose is to handle certain aspects of the container's lifecycle and state management.

Putting these together, shim-runc in containerd, refers to the shim layer that interfaces with runc, the lower-level container runtime. It helps in managing container processes in a way that enhances the overall functionality and stability of the container ecosystem. For instance, it allows containerd to restart without interrupting running containers, manage resource limits, facilitate better isolation and resource management, and handle logging and monitoring at a granular level.

Overall, this design where shim acts as an intermediary between containerd and runc ensures that the container runtime does not directly interact with the container processes, providing a cleaner, more modular architecture that enhances maintainability and reliability for container management, aligning with the industry standards set by the OCI.

In the following sections we emphasize on the procedures for building confidential containers (Section 3.2.1), the CONNECT nodes that are controlled by the K3s orchestrator (Section 3.2.2), as well as on the description of a simple example of a service/container instantiation (Section 3.2.3).

3.2.1 Building the CONNECT (K3s) Confidential Containers

The first step toward enabling K3s to create and manage confidential containers is the deployment of Enclave-CC in the K3s cluster. During this operation, shim-runc and containerd binaries are copied into K3s nodes, while the needed images (Agent enclave container and the Boot emty enclave container) are added in the image registry. In addition, containerd is configured to enable Enclave-CC run-time [27].

Then the user can deploy a new application (e.g. CCAM application) in the K3s cluster. For this, the user needs to define the Pod configuration in a set of manifest files. When the user makes a new request to the K3s APIServer, this request is forwarded to the shim-runc tool located in the containerd module. Then the shim-runc tool creates the Agent enclave container and requests from this container to pull the encrypted container image of the required application from the image registry.

When the Agent enclave container completes the download of the encrypted application image it needs to verify it. In this direction, the Agent enclave performs a remote attestation with the trusted Key Broker Service, which verifies the identity and trustworthiness of the Agent enclave and confirms the correctness of the downloaded image. In addition, the encryption key of the image is delivered. Then Agent enclave decrypts the received application image, encrypts again using a randomly-generated local key and based on the image it constructs a new encrypted file system.

In parallel, shim-runc creates the application enclave container and starts the Gramine instance (see details on the Gramine OS library in D4.2), which performs the initial enclave boot-up. When this process is completed, the application enclave container receives a local attestation request from the Agent enclave. During this local attestation process, the application enclave container receives the encrypted file system, moves it inside the enclave, decrypts it and installs it as the Gramine file system. Therefore, the application can be finally executed inside Gramine.

3.2.2 K3s in CONNECT: Cloud, MEC and vehicle nodes

As explained in the previous paragraphs the CONNECT orchestrator will be developed on top the Kubernetes container orchestration platform and specifically on top of its lightweight distribution called K3s. In Figure 3.3 the overall CONNECT orchestration solution is presented.

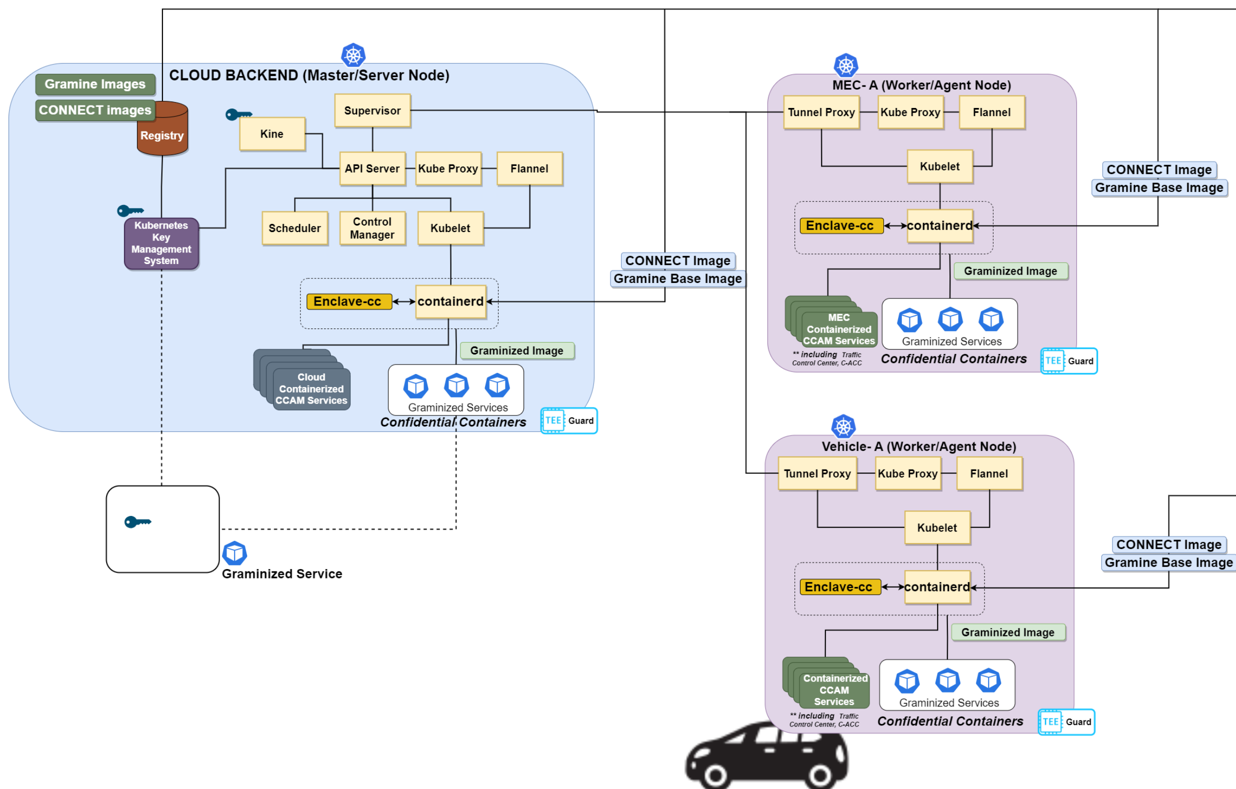

Figure 3.3: CONNECT orchestration solution

As illustrated in Figure 3.3, the CONNECT platform involves a Cloud node which serves as the Master/server node and a set of worker/agent nodes located on the MEC and in vehicles. In

practice each vehicle includes one worker node, while each MEC may support one or several worker nodes based on the actual requirements. Figure 3.3 presents the case which includes one Cloud node (Cloud Backend), one MEC node (MEC-A) and one Vehicle node (Vehicle-A).

In the ideal scenario we assume that the vehicle worker node is equipped with capabilities for supporting HW-protected application containers. In that case, the vehicle node is like the worker node in the MEC. If this assumption needs to be relaxed, we resort to software-based trusted execution environments (e.g., Gramine OS) for the vehicle node.

The main innovations of the CONNECT orchestration solution are: a) the realisation of the actual deployment of confidential containers in an operational (automotive) environment; b) the performance evaluation of actual automotive applications (e.g. CCAM) in the CONNECT environment. Regarding the latter, a set of evaluation scenarios are under discussion. These scenarios will be executed, while the results will be analysed and presented in the deliverable D5.3.

It is important to stress that at installation, K3s generates a self-signed CA to set up identities for each component in the cluster (as illustrated in Figure 3.4). This CA is then used to sign other certificates used within the cluster. We can discriminate between Server certificates and Client certificates described below:

- Server certificates: K3s generates a certificate for the server, which is used by the Kubernetes API server (this is signed by the CA). The API server certificate ensures secure communication within the cluster and for any external access to the cluster's API.
- Client certificates: K3s also generates various client certificates that are used by different K3s components to securely communicate with the API server. For example, the kubelet, the primary node agent running on each node, receives a client certificate to authenticate itself to the API server.

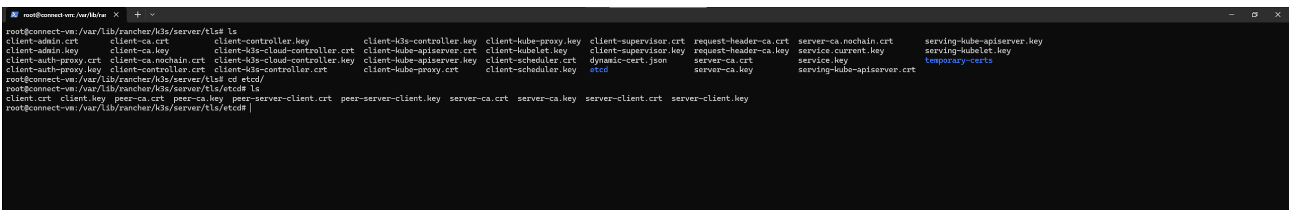

Figure 3.4: K3s generation of self-signed CA

In the rest of this subsection the main K3s components of the presented CONNECT orchestration solution are presented in detail. The figure and the description of certain components presented in the subsection is necessarily the same to that presented in D4.2.

API server

The API server is the interface through which all the operations and communications within a Kubernetes cluster are conducted. The API provides the mechanisms for creating, modifying, managing, and deleting resources (e.g., Pods, services, volumes and namespaces). It defines a set of RESTful endpoints for each type of resource, allowing for CRUD (Create, Read, Update, Delete) operations.

In this regard, each component in a Kubernetes cluster, including the Scheduler, Kubelet, Controller Manager, and others, acts as an HTTP client to the API server. They use standard HTTP

methods to query, update, create, and delete Kubernetes resources, thereby managing the state and operation of the cluster

Additionally, for watching resources (like the Scheduler watching for new Pods), Kubernetes often uses a mechanism like WebSockets or long polling with HTTP GET requests. This allows the Scheduler to maintain a persistent connection to receive real-time updates about the state of resources.

Kubelet

Kubelet is a kubernetes component that is present in all nodes and facilitates multiple interactions between master nodes and work nodes. Via the API servers and kubelet, nodes are registered (or deregistered) to the cluster as worker nodes. This process includes reporting various details about the node, like its resources (CPU, Storage, memory, etc.), supported features, and health status. Specifications of services and deployments are passed to kubelet via the API server. These specifications describe the containers that need to be run, along with their configurations and interactions with other Pods and services.

In coordination with the container runtime (e.g., containerd, docker, cri-o) kubelet manages the lifecycle of Pods (such as instantiation, termination, modification) and containers within them, whereas in conjunction with the network plugin (e.g., flannel, calico, etc.), kubelet sets up and manages the network for the Pods, ensuring they can communicate with each other and with other components in the cluster (within the same or different nodes).

Finally, kubelet also monitors the resource usage of containers (and of the nodes that host them) and reports this information back to the API server. This data is used for scheduling decisions, to ensure that resource limits are enforced, to perform various health checks as well as to have deployments and services running at their desired state, i.e., as intended and defined by the service manifest. The described functionalities (along with the procedure for building confidential containers, see Sec. 3.2.1) realize the relevant lifecycle management (LCM) operations of CONNECT containerized applications.

Scheduler

The scheduler is responsible for assigning Pods to nodes. It makes these decisions based on several factors such as resource availability (this includes CPU/GPU, memory, Storage) and other custom requirements, e.g., network interface card types, constraints specified in the Pod configuration (like node selectors and affinity/anti-affinity rules), and other policies (e.g., load balancing at the host level). It works closely with the API server and considers a variety of factors and policies to maintain the desired state and performance of the cluster. In brief, it operates as the resource orchestrator (RO) of the cluster (see the end-to-end architecture of the CONNECT K8s cluster at D2.1), selecting nodes capable of hosting the workload of containerized applications, taking also into account the cluster policies and constraints.

Controller Manager

The Controller Manager runs various controllers, each responsible for specific aspects of cluster management. These include the Node Controller, Job Controller, Endpoints Controller, Service Account and Token Controllers, and others. These controllers watch the state of the cluster through the Kubernetes API and make changes attempting to move the current state towards the desired state. For instance, if the manifest of the service dictates that the deployment should be composed of two Pods (e.g., replicaset 2), and one of the Pods fails, the controller manager will detect the failure and try to deploy a new Pod to reach the desired state of the deployment, i.e., 2 Pods.

To detect such changes, maintain the desired state, and remedy cases of failure, the Controller Manager works primarily with the Kubernetes API Server continuously watching for changes in various objects, and works to reconcile any differences between the observed state and the desired state. The API Server, in turn, interfaces with Kine (cf. next subsection) to store or retrieve this state information from the backend database service.

Kube-proxy

The primary function of kube-proxy is to maintain network rules on nodes and it works in coordination with Kubernetes network plugins such as flannel, calico, etc. These rules allow network communication to Pods from network sessions inside or outside of the cluster. We can abstract it in general as a routing table, with mechanism to monitor and oversee these rules and changes. Hence, these rules allow network traffic to be directed to the appropriate Pods based on the service IP and port.

Kube-proxy also performs basic load balancing. When a service is created to represent a set of Pods, kube-proxy sets up routes so that the service IP is directed to one of these Pods. It can distribute network traffic to Pods using round-robin and other routing methods.

Flannel

Flannel is a container network interface (CNI) plugin. When a Pod is created, the Kubelet works with the network plugins (like flannel, calico, etc.) to allocate a network interface for the Pod, assign IP addresses, and set up the necessary routes so that Pods can communicate with each other.

Kine in K3s

Kine is a datastore shim that allows etcd to be replaced with other databases. K3s supports MySQL, PostgreSQL, SQLite, and more, for storing cluster data. It acts as a translation layer that allows K3s to use these databases, instead of etcd that is used in full kubernetes clusters. Generally, it provides storage backend flexibility, helping to make Kubernetes accessible and practical for a broader range of environments and use cases, especially where resources are limited or simplicity is a priority. It is important to note that Kine translates and structures the data in a way that is compatible with the Kubernetes API's expectations. This ensures that, regardless of the underlying database technology, the data is managed in a consistent and reliable manner, as per Kubernetes standards.

Kine database system accommodates all cluster state data. This encompasses all information about the current state of the cluster. It includes details about Kubernetes resources such as Pods, services, deployments, statefulsets, and daemonsets, as well as information about the state of the worker nodes (resource availability, health status etc.). It also stores information about users, roles, and role bindings, which are crucial for access control and permissions within the cluster, as well as data related to the scheduling and orchestration of containers, such as Pod status (e.g., crushed, pending, available), events, and logs.

Such data are essential for the Kubernetes master node (control plane) to manage and orchestrate the cluster. Via Kine and the API server, all other Kubernetes components (kubelet, scheduler, controller manager) retrieve information with respect to the cluster state, and perform their associated functionalities. Although Kine itself doesn't manage keys, it plays a role in storing them. In this regard Kubernetes secrets (applying encryption of secrets at rest) that Kine stores include passwords, OAuth tokens, SSH keys and service account tokens. The latter are used for authenticating applications or users for accessing the Kubernetes API.

Tunnel proxy

One of the primary roles of the tunnel proxy in K3s is to enable communication between the control plane (server/master nodes) and the worker/agent nodes. It facilitates secure and efficient communication within the cluster, especially in environments where direct network connections between nodes are restricted (behind NAT or in different network segments, etc.). It basically tunnels traffic from control and worker nodes (facilitating the communication between all Kubernetes components). It is not a mandatory K3s element, but it adds to extra security. By tunneling traffic, it can help ensure that the data transmitted across the cluster is encrypted and protected from external threats, which is vital in maintaining the overall security of the cluster.

Supervisor in K3s

In K3s the supervisor is in close collaboration with the API-server, and these two components can be collocated (same host), or not. Essentially the supervisor functions as a relay of information (e.g., a load balancer) between the K3s agents (worker nodes) and the control plane functionalities of the K3s server (master node). The information is exchanged between the Tunnel Proxy of each agent and the supervisor of the server. In addition, the supervisor plays the role of an added layer which acts as an intermediate firewall for enhancing security for in-cluster communications.

At the initial phase, a websocket connection is created in order for worker nodes to initiate the node/agent registration process. Subsequently, K3s agents connect to the supervisor (and kube-apiserver) via the local load-balancer, retrieving a list of available service connect endpoints. The agent retrieves a list of kube-apiserver addresses from the Kubernetes service endpoint list in the default namespace. Those endpoints are added to the load balancer, which then maintains stable connections to all servers in the cluster (high availability cluster), providing a connection to the kube-apiserver that tolerates outages of individual servers.

3.2.3 CONNECT Confidential Containers Flow

In Figure 3.5 the whole process for the instantiation of a service (e.g. CCAM application) is presented. As explained in the previous section, this container is not regular, but it is enhanced to be a confidential container by utilising the capabilities of Gramine and Enclave-CC. The steps of the flow, which are described in detail below, they are in line with the K3s details provided previously and the approach described in D4.2.

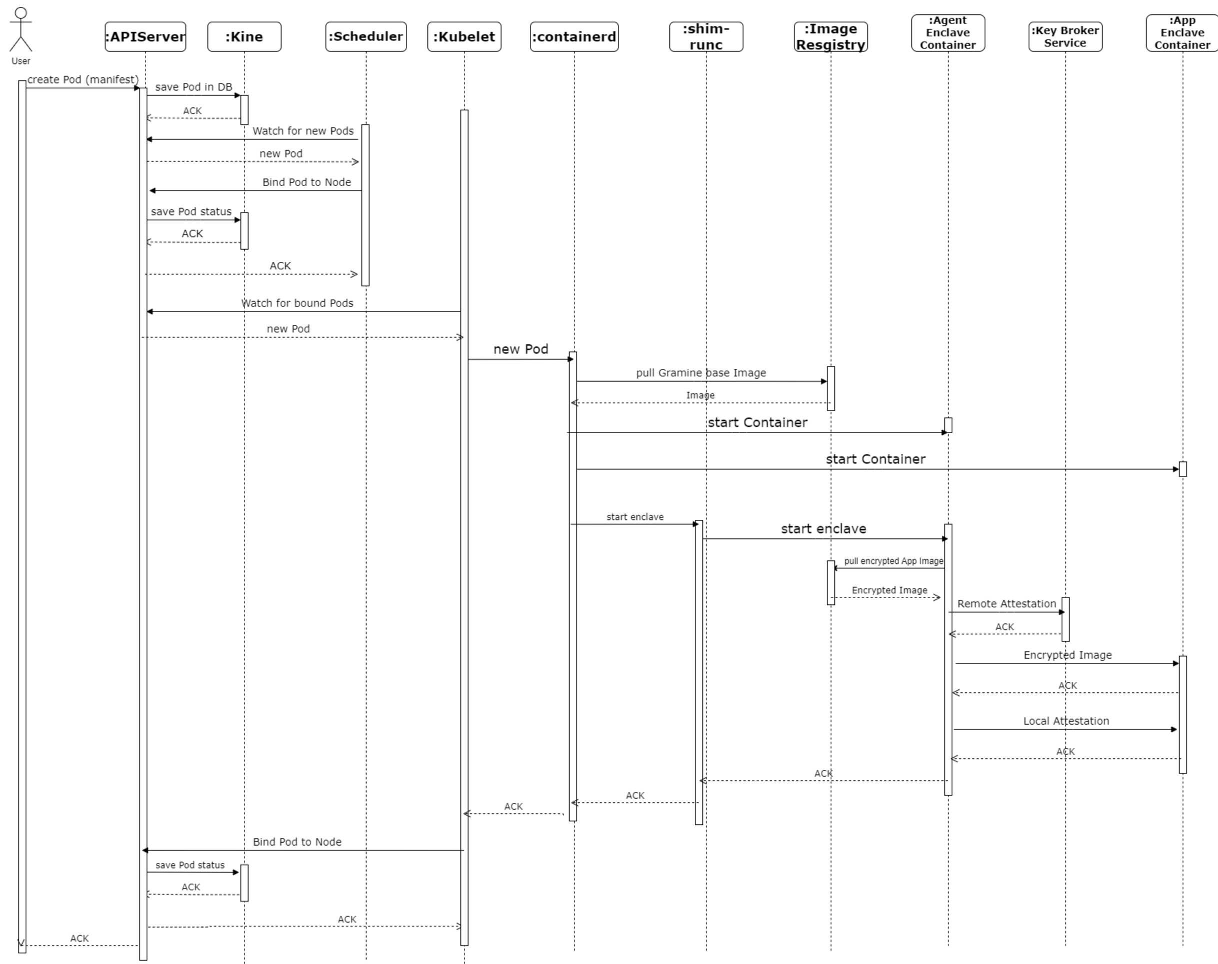

Figure 3.5: Service instantiation in confidential containers

The process of instantiating confidential containers in CONNECT involves multiple components interacting with each other, for the deployment of the service workloads at particular node(s) within the cluster (e.g., to the CONNECT MEC nodes), to apply the Enclave-CC approach and execute application inside a Gramine environment. The process begins with a user (or an automated system) creating a deployment through the K3s API server. This is typically done using kubectl, the Kubernetes command-line tool, to submit a YAML file (or a set of manifest files) that describes the desired state of container services, or via a kubernetes package manager such as helm3, that helps streamline the deployment and management of Kubernetes applications.

The API server submits the new deployment to Kine (i.e., the translation layer between Kubernetes and the underlying database), along with its status (e.g., new assignment, pending for deployment). The scheduler constantly watches the API server for new work items (Pods that need to be assigned to a cluster node). Once a new assignment is found, the scheduler queries the API server to gather current data about the state of the cluster. This includes information about the resources available on each node, such as CPU, memory and storage, affinities, anti-affinities, taints, tolerations and other scheduling constraints that are defined in the Pod's specification or the cluster's policies. The scheduler uses this information, as the K8s resource orchestrator (RO), to evaluate which nodes have sufficient resources to run the Pod. Once a node is selected, the RO communicates back to the API server to bind the Pod to the chosen node, and API server updates the Pod's information in Kine to reflect this binding.

Through the tunnel proxy, the kubelet agents at each worker node are continuously watching the

API server for Pod bindings. Once a kubelet detects that a new Pod assignment is destined for its node, it interacts with the container runtime engine (e.g., docker, containerd, cri-o) to prepare for container deployment. Particularly, Kubelet passes the container image, environment variables, volumes, and other necessary information (based on the K3s service manifest) to the container runtime, e.g., containerd. Subsequently, containerd pulls the container image from the image registry (or uses a local copy if available/instructed). After the image is pulled, containerd creates the container based on the specifications provided by the service manifest files.

Additionally, the kube-proxy and the network plugin (e.g., flannel) create the necessary virtual networks to establish communication between different Pods (potentially across hosts) or for Pod communication with services outside the cluster.

As explained in section 3.2.1, containerd utilises the functionalities of shim-runc tool. The shim-runc tool creates the Agent enclave container and application enclave container and starts the Gramine instance, which performs the initial enclave boot-up (see Figure 3.5). The shim-runc tool requests from Agent enclave container to pull the encrypted container image of the required application from the image registry. Then, the Agent enclave performs a remote attestation with the trusted Key Broker Service to validated the received image. The Agent enclave based on the image it constructs a new encrypted file system. Finally, the application enclave container receives a local attestation request from the Agent enclave and receives the encrypted file system to be installed in the Gramine file system. Therefore, the application is executed inside the confidential container.

Once the confidential container is running, containerd continues to manage its lifecycle and reports the status back to the kubelet. The kubelet updates the status of the Pod in the API server, which in turn updates the overall deployment status in Kine. This status information is also constantly monitored by the controller manager which performs automated health checks and adjustments when needed, to ensure that the desired state declared by the user in the deployment phase, is achieved and maintained.

3.3 CONNECT Orchestrator in Lab Environment

In the preceding sections, we outlined a comprehensive architecture based on K3s, and and array of components necessary to meet CONNECT's objectives. It's important to note that a significant portion of this architecture and the associated services have undergone rigorous testing within a controlled laboratory environment at the ICCS premises. Particularly, we have an operational K3s cluster, including a master node and a set of worker nodes were we build confidential containers (as described in Section 3.2.1), deploy them in relevant worker nodes, validate and benchmark the relevant RO and LCM operations.

To facilitate the cellular connectivity between the worker nodes and the master node, we utilize the ICCS 5G testbed, and particularly Amarisoft[2] solution for 5G connectivity. This setup allows the creation of a K3s cluster under real operating conditions, with 5G connectivity for testing all relevant operations. Figures 3.6 and 3.7 depict the 5G New Radio (NR) basestation, the worker node that hosts CONNECT services and application workloads, and the master node that resides at the ICCS servers. The interconnection of the worker node towards the master node (i.e., the 5G link) is facilitated via Teltonika's RUTX50 Industrial 5G router[3], also depicted in Figure 3.6.

[2]https://www.amarisoft.com/

[3]https://teltonika-networks.com/products/routers/rutx50

Hence, the K3s worker node is connected to the 5G network, which is sequentially connected via an Ethernet connection to the K3s master node, residing at the ICCS rack server depicted in 3.7

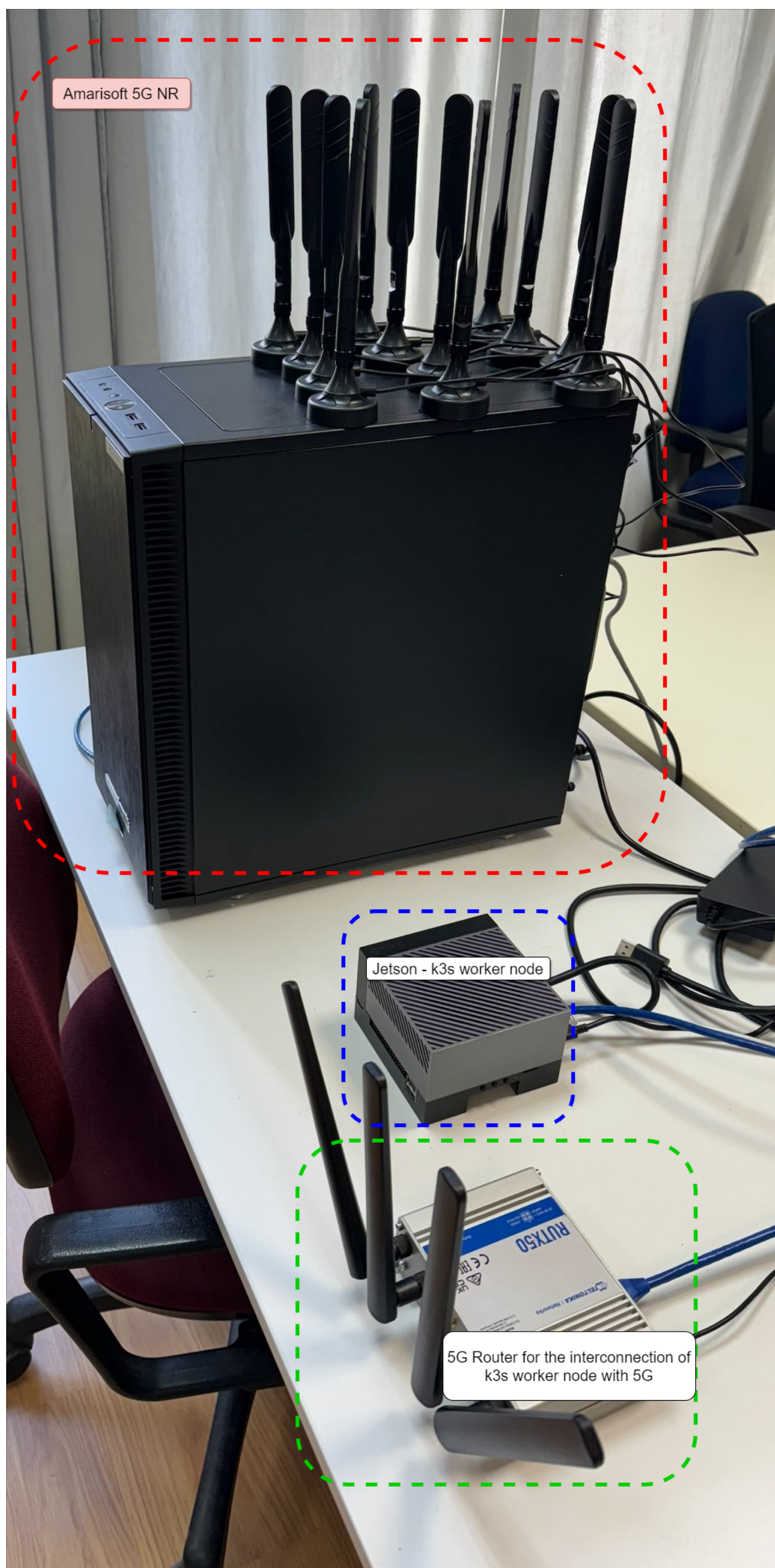

Figure 3.6: K3s worker node and Private 5G network at ICCS premises.

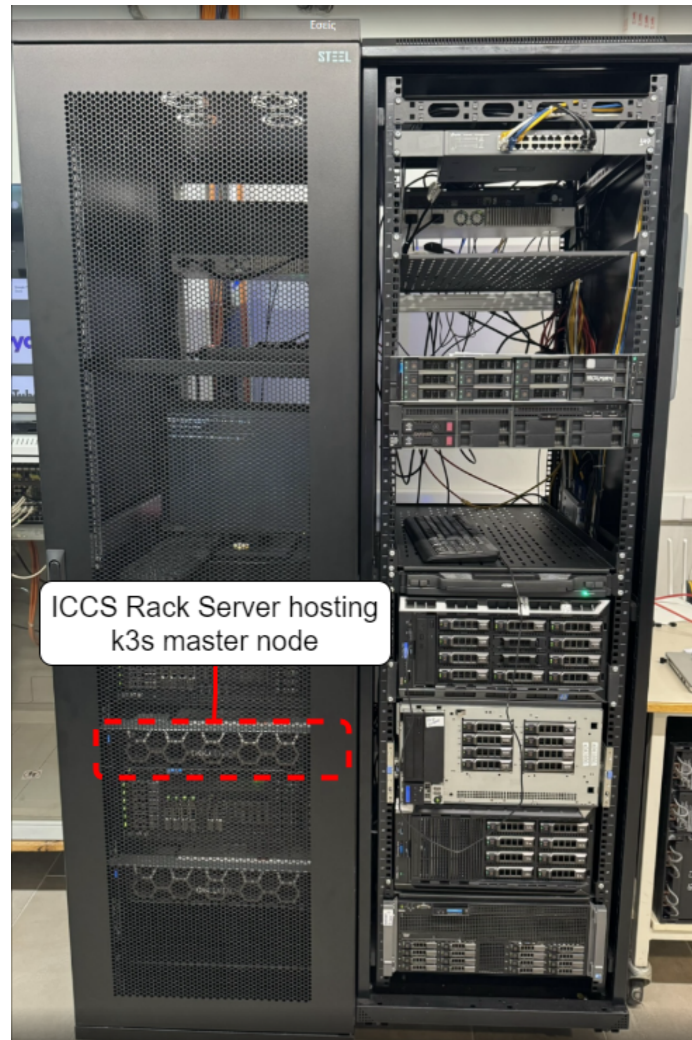

Figure 3.7: K3s master node hosted at ICCS servers.

The local setup at the ICCS 5G testbed serves as a first step towards identifying and addressing potential issues, ensuring that the discussed architecture and the CONNECT view is practically viable and ready for broader implementation and WP6 integration and real-world demonstration (see SMTD use-case in D2.1).

Figure 3.8 showcases the control plane functionalities of the the K3s master node and the relevant K3s services at the ICCS testbed, based on the kubectl commandline tool.

Figure 3.8: K3s master node services for LCM support of cluster operations.

In our setup, we perform various preliminary tests to benchmark the computation tasks on the vehicle, i.e., the $T1$ time presented in Section 2 which refers to the (CCAM) service processing and trust computation delay. Similarly, the ICCS rack servers (added as K3s worker nodes) will serve as the testing ground for benchmarking $T3$ and the computation at the infrastructure side (see Section 2). This on-going benchmark will not only provide us with a detailed understanding of our system's capabilities and limitations (focusing on CONNECT services) but has also established a solid foundation for future developments. Importantly, the insights garnered from this benchmark will be instrumental in the CONNECT task offloading service (see Section 2 and Section 3.4).

Other features so far tested at the ICCS 5G testbed, is the potential overhead of LCM operations (e.g., creation, instantiation, modification, termination) on graminized CONNECT confidential containers (see Deliverable 4.2, Section 5) and their easy-to-use and seamless integration with K3s orchestrator.

Building upon the successful laboratory testing, the next step in our project's progression involves the integration of the K3s systems (worker and master node) at the pilot site, where the demonstrator phase is set to occur. The K3s cluster will be further tasked to orchestrate various CONNECT components (e.g., TAF, CCAM service containers, confidential containers, etc.) and demonstrate part of the CONNECT vision. Hence, the SMTD use-case pilot site will serve as a proving ground, allowing us to fine-tune the systems based on practical experience gained from in-lab experimentation in order to efficiently support the forthcoming WP6 (SMTD) demonstrator.

3.4 Towards the CONNECT task-offloading realisation

The aforementioned CONNECT technology will find direct application to the Slow Moving Traffic Detection (SMTD) use-case (see D2.1). More importantly, the SMTD use-case will be realised and demonstrated in a real-world road environment.

As part of this use-case, the task offloading capability, secured by the CONNECT mechanisms and highlighted in Section 2.3 (as open research questions for experimentation), will be shown. Towards that end, the project will need to introduce in the coming D6.1:

- an offloading scenario to be coherently coupled with the SMTD use-case semantics. The scenario should allow the project realise and experimentally evaluate the involved trade-offs (i.e., basically the impact of the CONNECT TAF/trust mechanisms deployed in confidential containers, over the involved end-to-end service delays)
- the aforementioned introduced task-offloading scenario should be integrated into/ supported by the CONNECT vehicle platform. Potential needs for dedicated HW/SW set-up, closely related to the SMTD case, should be also fulfilled.
- finally, the aforementioned technology for the deployment of confidential containers should be complemented with sensing/telemetry tools which would collect all required data to facilitate the relevant offloading decisions (see Section 2.3)

All the above directions will be introduced and clearly presented in the 'Use Case Analysis' part of the D6.1.

Chapter 4

The CONNECT Blockchain Infrastructure for Auditability

4.1 The Blockchain motivation in CONNECT

The adoption of the Distributed Ledger Technology (DLT) in the CONNECT system is crucial, since it brings forward in a decentralized manner, trusted and secure data operations that ensure the auditability and trustworthiness of the wide range of transactions performed in CONNECT, hence enhancing its reputation [25]. The CONNECT DLT enables the storage of critical data such as trust models and attestation evidence within the CONNECT system, while supporting access policies that restrict access to the stored information.

The intrinsic decentralized and secure nature of DLT not only ensures auditability, but also facilitates the monitoring of trustworthiness evidence, particularly from the attestation processes, resulting in the establishment of a historical trust record for any component participating in the CONNECT system. Hence, any authenticated and authorized entity is able to verify the current state of a component. Moreover, by leveraging the historical trust data stored on the DLT, further sophisticated trust assessments can be performed towards establishing a reputation system in CONNECT. By prohibiting access to unauthorized entities through the application of ABAC (Attribute-Based Access Control) and ABE (Attribute-Based Encryption) mechanisms (see also Section 5), the CONNECT DLT safeguards the calculation of RTLs and ATLs in the system and ensures that they will not be tampered with, therefore boosting the security and trustfulness of operations in CONNECT.

The importance of CONNECT DLT lies on the need for provenance, as the attestation evidence stored in the Blockchain legders is essential information that is consumed by the OEM, which processes the attestation raw traces towards discovering new vulnerabilities in the CONNECT system. Furthermore, in the context of trust assessment processes, the CONNECT DLT can provide a kind of mechanism for the emulation of a trust system, as it stores the trust opinions that can be used for the establishment of reputation in CONNECT components. It has to be mentioned that although theoretically there could be other means of storing attestation evidence and trust models in CONNECT, the fact that in CONNECT system such data may be needed by various sources like CCAM certificates, enhances the need for the application of DLT that ensures security and trustfulness of stored data along with strong authentication and authorization mechanisms.

4.2 The Blockchain architecture and the CONNECT approach

4.2.1 Overview on Blockchain technologies for CONNECT needs

In order to decide upon the most suitable BC technology for the needs of CONNECT, a state-of-the-art review is presented below. A focus has been given on Hyperledger Fabric, Hyperledger Besu and Ethereum as they are the best candidates based on the following CONNECT DLT design criteria: (a) scalability, (b) strong authentication and access control, (c) extendability.

Hyperledger Fabric

This is a permissioned Blockchain platform developed by the Linux Foundation's Hyperledger project, designed for enterprise use cases, providing a modular and customizable architecture. Hyperledger Fabric (HLF) supports various consensus mechanisms, including Practical Byzantine Fault Tolerance (PBFT). This flexibility allows organizations to choose the consensus algorithm that best fits their requirements. HLF supports smart contracts known as chaincode, which can be written in languages such as Go or Java. HLF allows for fine-grained control over permissions and privacy, where participants can be granted specific access rights to data and transactions [6].

Hyperledger Besu

This is another project under the Hyperledger umbrella, specifically developed for enterprise Blockchain applications. It is an Ethereum client that supports both public and private network deployments. Hyperledger Besu supports various consensus mechanisms, such as Proof of Work (PoW) and Proof of Authority (PoA). Hyperledger Besu is Ethereum Virtual Machine (EVM) compatible, which means it can run Ethereum smart contracts. This feature makes it interoperable with existing Ethereum tools and applications. Like HLF, Hyperledger Besu allows for privacy and permissioning features. It enables organizations to set up private networks with controlled access to participants [11].

Ethereum

This is an open-source, decentralized and permissionless Blockchain platform that enables the creation and execution of smart contracts and decentralized applications (DApps), officially launched in 2015. Ethereum uses a Proof of Stake (PoS) consensus mechanism called the Beacon Chain for its Ethereum 2.0 upgrade, enabling scalability, security, and energy efficiency. Ethereum has introduced the concept of smart contracts, having the terms of the agreement directly written into code. Solidity is the primary programming language for Ethereum smart contracts [23].

Selection of HLF for CONNECT DLT

As the nature of the CONNECT system demands a very strong level of security and trust, a permissioned Blockchain technology is the obvious choice. Between the two Hyperledger projects, HLF has been selected in CONNECT for several reasons [13]. Primarily, HLF is well-recognised for its stronger access control compared to Besu, having more robust certificate and identity management mechanisms. More specifically, unlike Besu, HLF has its own Certification Authority. In terms of modularity, HLF is designed with a modular and extensible architecture that allows more flexibility in customizing the Blockchain network over Besu. Also, as Besu supports primarily the PoW consensus algorithm, which is considered less energy-efficient than some other consensus mechanisms, especially for private or consortium networks, HLF is the better choice. Regarding privacy, HLF provides a more structured way to manage privacy and confidentiality in multi-party scenarios. For smart contracts language, Hyperledger Besu primarily supports Solidity, while the

multilingual support in HLF is an advantage for CONNECT development and integration activities. Moreover, HLF's exclusive focus on permissioned networks makes it more straightforward with specific privacy and governance requirements. Nevertheless, it has to be mentioned that CONNECT can be agnostic to Blockchain Ledgers and as such it is not tightly linked with HLF, but HLF has been an excellent selection for instantiating the schemes for better authentication, authorization and integration of cryptoprimitive mechanisms in CONNECT.

4.2.2 Architecture

As described in detail in D2.1 [10], the flow of actions in the CONNECT DLT includes two phases, namely the setup and the runtime. During the setup phase, the trust models, RTLs and trust policies coming from the TAM are deployed onto the DLT, as well as the registration of the TAF Agents - both on the MEC and the vehicle sides - is performed to get automatically notified in case of upgrades. These trust models, policies and RTLs that reside within the Private Ledger, are only accessible to authorized entities through the operation of the Blockchain Peers. During the runtime phase, failed attestation evidence is recorded onto the DLT, either for the MEC or the CCAM services and the vulnerabilities are identified by the corresponding security administrators and OEMs. Upon recognition of new vulnerabilities, the TAM with the assistance of RA (Risk Assessment), recalculates the RTLs. Within the CONNECT DLT, no personally identifiable information (PII), such as location or other sensitive data, are stored. Only low-level system traces derived from the attestation processes are stored. As these traces contain sensitive information, Attribute-based Encryption (ABE) and Attribute-based Access Control (ABAC) ensure that non-authorised users will not access such information, while data sovereignty is further boosted utilizing Self Sovereign Identity (SSI) concepts. The CONNECT DLT comprises the following internal components, as depicted in Figure 4.1:

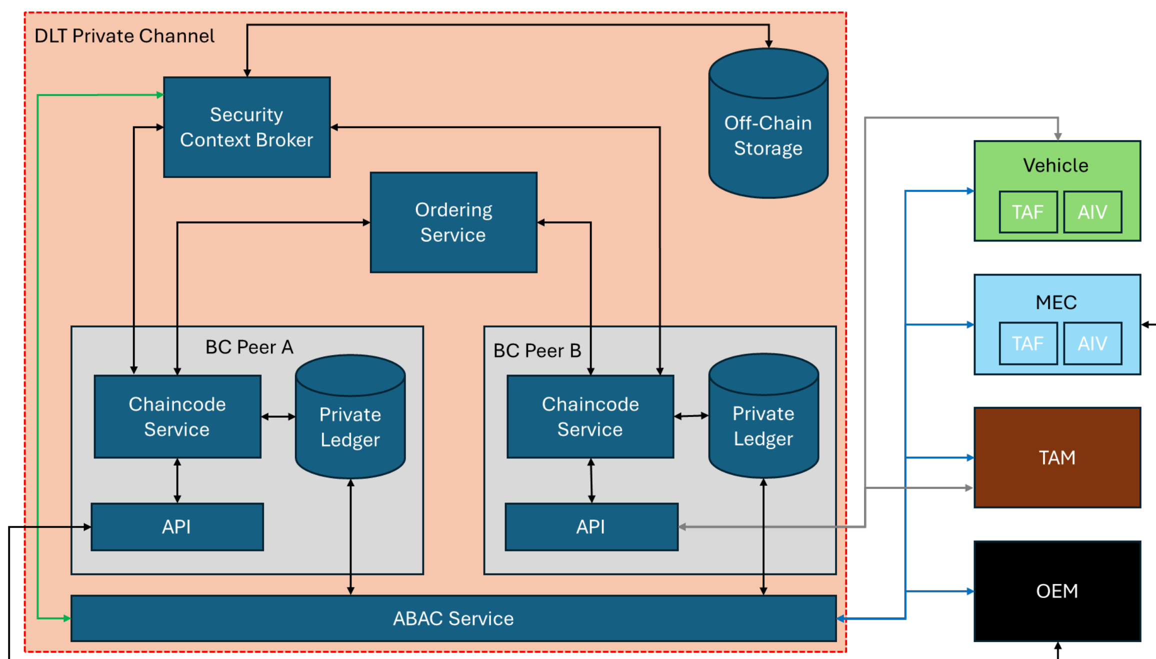

Figure 4.1: CONNECT DLT Architecture

Blockchain Peer

A Blockchain Peer represents an essential component within the CONNECT DLT network, as it hosts the chaincode responsible for executing smart contract functions and manages the ledger where transaction information is recorded. A BC Peer communicates with the TAM, OEM and TAF agents (MEC and vehicle sides) for the secure recording and retrieval of information, such as trust models and attestation evidence in the CONNECT Private Ledger. This collaboration supports the execution of smart contracts that are written in DLT. The BC Peers, which run independently of each other and maintain their own copy of the Private Ledger, play a pivotal role in the CONNECT DLT, serving as integral components that conduct operations, maintain ledgers and actively participate in consensus schemes. These schemes are crucial for processing transactions and securely incorporating them into the Blockchain. In CONNECT, the design choice is to not have strictly one BC Peer per vehicle, but to maintain a suitable number of BC Peers that will enable the smooth operation of DLT processes. The optimal number of BC peers for supporting a variety of vehicles will be documented in the deliverables D6.1 and D6.2 under WP6 evaluation activities.

Chaincode Service

The Chaincode Service comprises the execution engine responsible for the various smart contracts deployed across different peers that perform the data transactions in CONNECT DLT. The Chaincode Component handles operations directed by the Security Context Broker regarding the creation and deployment of smart contracts containing attestation evidence and trust models/policies. The API layer of the different peers transfer the input to the chaincode service.

Ordering Service

The Ordering Service is employed in CONNECT DLT to coordinate transactions sequentially and construct new blocks in the Blockchain. When a transaction (e.g. storage of data in the Private Ledger) is transmitted by a CONNECT component (e.g. MEC) to a peer via the API layer, the peer validates and executes the transaction and returns the results to that component. The peer subsequently submits another transaction containing these results to an Ordering Service. As numerous transactions are sent to the Ordering Service, it aggregates them into blocks. Given that there can be multiple Ordering Services, each receiving transactions from different peers, collaboration is established between them to organize transactions and formulate them into blocks of the ledger. Consensus algorithms ensure unanimous agreement on the exact order of transaction blocks. Once consensus is achieved, a distinct order is established and transactions are consolidated into blocks, which are stored onto the ledgers of the peers.

Security Context Broker

The Security Context Broker (SCB) comprises a trusted intermediary connecting the Blockchain network with the rest of the CONNECT system (see also the flow in Figure 4.4). Its main responsibility lies in overseeing the creation and deployment of smart contracts on the ledger. The SCB translates the data transactions, as received by the peers, into smart contract logic and subsequently deploys the smart contracts onto the ledger. Moreover, access to and querying of specific attestation evidences or trust models/policies are restricted by the SCB to components possessing the necessary attributes, through the application of Attribute-based Access Control (ABAC) mechanisms (see also Section 5.1). Apart from its interaction with the peers, the SCB also communicates with the off-chain storage of CONNECT DLT for storing and retrieval of actual transaction data that are not stored in the Private Ledger due to their sensitive nature or large size.

API Layer

The API layer enables the communication and interaction between external entities (e.g., CON-

NECT components, OEM) and the DLT (see also the flow in Figure 4.4). The API Layer exposes the interfaces that are invoked by external entities, guiding the communication between the BC Peers and the SCB. Through the utilization of the API layer, the integration with smart contracts is enabled for the facilitation of transactions and the recording/retrieval of data. Moreover, the API layer contributes to the finalization of transactions before they are forwarded to the Ordering Service, by facilitating the organization of essential transaction details prior their submission to the Ordering Service towards their storage in CONNECT DLT.

Off-chain Storage

The Off-chain Storage comprises the storage of ABE-encrypted data that cannot be stored directly on the DLT network due to their sensitive nature or large size, while their corresponding pointers are stored in the Private Ledger maintaining the location of the stored data. These pointers are encrypted with ABE mechanisms for safeguarding security of information in CONNECT, as only authorised access is allowed to CONNECT components, such as MEC, through interaction with the Security Context Broker and application of ABAC policies.

Private Ledger

The Private Ledger stores the hashes of the smart contracts that have been executed regarding the recording/retrieval of attestation evidence or trust model/policy, and the corresponding pointers to the encrypted off-chain data to ensure the fast performance of the ledger. The hashes are provided by the CONNECT components as input to specific smart contracts through the API Layer. The pointers to the data stored in the Off-chain Storage are provided by the Security Context Broker that facilitates the storage of the encrypted data to the database of CONNECT DLT.

4.3 Blockchain data requirements

Based on the motivation exploiting the Blockchain technology in CONNECT, a set of dedicated requirements has been defined towards the design of CONNECT DLT and its subsequent development in the next phase of the project.

These requirements facilitate the lifecycle of the two distinct data flows in CONNECT: (a) the management of attestation data and (b) the management of trust policies/models and RTLs. More specifically, at first, a general group of requirements that cover the full data lifecycle of CONNECT DLT operations is described in Table 4.1.

Then, different sets of specific requirements focusing on certain aspects of CONNECT DLT are defined, delving into further details in DLT features, namely:

- Confidentiality and integrity: This aspect is related with the CONNECT DLT mechanisms for secure storage and immutability of information (Table 4.2),
- Authorisation and access control: This aspect refers to the leverage of Verifiable Credentials and access control mechanisms in CONNECT DLT as also mentioned in Section 5.1 (Table 4.3),
- Encryption: This aspect refers to the application of ABE mechanisms in CONNECT DLT as also mentioned in Section 5.2 (Table 4.4) and

- Non-repudiation and accountability: This aspect is related to the establishment of CONNECT DLT as a reputation system owning information necessary for the certification of any CONNECT entity (Table 4.5).

Table 4.1: CONNECT DLT Requirements

Req#	CONNECT DLT Requirements
DLT-01	The CONNECT DLT shall include only private ledgers towards ensuring trustworthiness and security of information transacted within CONNECT.
DLT-02	The CONNECT DLT shall leverage ABAC mechanisms for granting access to the different CONNECT components, as a permissioned Blockchain network.
DLT-03	Attestation evidence, trust models, and trust policies shall be deployed as smart contracts onto the CONNECT DLT to facilitate the auditability and reputation of CONNECT system.
DLT-04	The CONNECT components shall acquire necessary VCs to obtain access to the CONNECT DLT.
DLT-05	The Security Context Broker shall undertake along with ABAC Service the access control policy to the CONNECT DLT.
DLT-06	The data sent from the CONNECT components to the CONNECT DLT shall be transformed into an appropriate format to be used for the deployment of smart contracts.
DLT-07	The data that will be stored in the Off-chain Storage shall be encrypted leveraging ABE mechanisms.
DLT-08	The Private Ledger of the CONNECT DLT shall contain only the hashes of attestation evidence, while the actual attestation data shall be stored in Off-chain Storage.
DLT-09	The storage location of data in Off-chain Storage shall be maintained as pointers in the Private Ledger.

4.4 Definition of Smart Contracts

4.4.1 Smart Contract deployment workflows

As also mentioned in previous subsections, attestation evidence, trust policies, trust models and RTLs are all stored in the CONNECT DLT through the creation and execution of smart contracts. Smart contracts are self-executing contracts directly written into chaincode. They operate on the CONNECT Blockchain infrastructure, enabling secure and automated execution of transactions, ensuring transparency and immutability. Smart contracts leverage a decentralized environment, by exploiting a network of peers that reach consensus on the state of the Blockchain, thus, eliminating the need for a centralized authority to control contract execution.

The smart contracts accommodate the logic that designates the storage and retrieval of data in the CONNECT DLT. The chaincode of the smart contracts contains the necessary attributes to facilitate the compatible processing by the Blockchain peers. The Security Context Broker commences the execution of the smart contract by sending as a transaction the chaincode along with the appropriate signing parameters to the Blockchain peer. Afterwards, the transaction is

Table 4.2: Confidentiality and Integrity Requirements

SEC1 Confidentiality and Integrity The data in CONNECT DLT should be safeguarded through appropriate measures to assure the integrity, confidentiality, and availability during the whole data lifecycle	
Req#	CONNECT DLT SEC Requirements
DLT-SEC-01	The CONNECT DLT shall be deployed onto such a number of Blockchain peers towards assuring the data integrity in the ledger.
DLT-SEC-02	The consensus mechanism of the CONNECT DLT shall ensure the data integrity in the ledger.
DLT-SEC-03	Smart contract mechanisms shall be used in the CONNECT DLT for the secure recording of data transactions in the CONNECT system.
DLT-SEC-04	The Security Context Broker is the only intermediary between the CONNECT components and the Off-chain Storage, ensuring security and confidentiality in data access.
DLT-SEC-05	The CONNECT DLT and the Security Context Broker shall exchange information in a secure manner.
DLT-SEC-06	The Security Context Broker shall govern the access of CONNECT components to the CONNECT DLT by exploiting ABAC functionalities.
DLT-SEC-07	Sensitive or large-size data will be stored in the Off-chain Storage of the CONNECT DLT to ensure efficiency and availability (e.g., sensitive raw traces with their deviceID).
DLT-SEC-08	The Private Ledger of the CONNECT DLT shall maintain as encrypted pointers the locations of data stored in the Off-chain Storage.
DLT-SEC-09	The CONNECT DLT shall exploit an Off-chain data storage component to safeguard data availability.
DLT-SEC-10	The Off-chain Storage of the CONNECT DLT shall maintain the stored data in an encrypted format to ensure security and confidentiality.

designated to a recipient peer, which performs a validation check on the transaction towards ensuring the correctness of the smart contract chaincode. This chaincode is written as a block in the Private Ledger and the state of the ledger is updated (Figure 4.2).

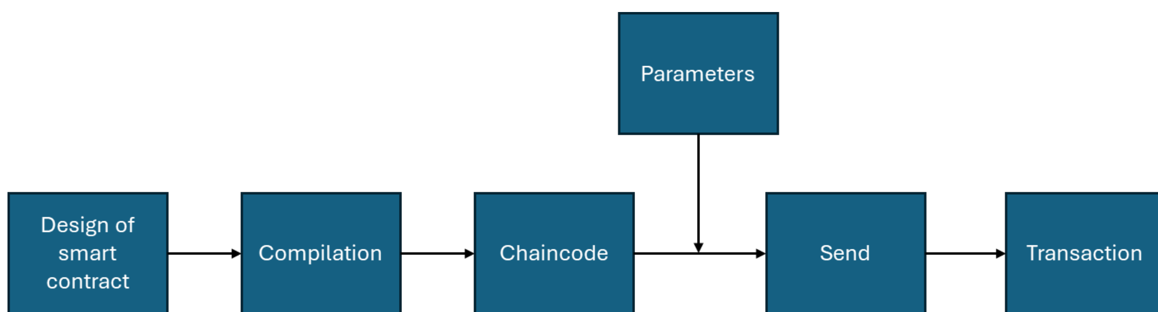

Figure 4.2: Smart contract transformation

Upon the transformation of a smart contract into transaction, the smart contract for that transaction can be executed by sending an execution request to the API Layer. This request is transformed into chaincode and sent to a Blockchain peer, which takes over the execution of the smart contract and converts it to chaincode that is sent to the Ordering Service. The received transac-

Table 4.3: Authorisation and Access Control requirements

SEC2 Authorisation and Access Control The CONNECT components (e.g., OEM, MEC) shall have access to the CONNECT DLT based on specific attributed based access policies, ensuring the authentication and authorisation mechanisms of CONNECT DLT.	
Req#	CONNECT DLT SEC Requirements
DLT-SEC-11	The access policies dictating the data transactions shall be contained in the deployed smart contracts.
DLT-SEC-12	The ABAC Service shall be an integral part of the Security Context Broker.
DLT-SEC-13	The data that are stored in the CONNECT DLT shall be accessed only by the CONNECT components that hold the required attributes.
DLT-SEC-14	The ABAC Service shall coordinate the corresponding access of the different CONNECT components to the CONNECT DLT.

Table 4.4: Encryption requirements

SEC3 Encryption Within CONNECT DLT, all data will be encrypted through ABE mechanisms towards safeguarding the privacy and security of data, both in the Private Ledger and the Off-chain Storage.	
Req#	CONNECT DLT SEC Requirements
DLT-SEC-15	The encryption mechanisms in the CONNECT DLT shall exploit ABE so that CONNECT components with the required attributes can decrypt the data.
DLT-SEC-16	The ABE mechanisms shall allow the encryption of data both in the Private Ledger (for locations of data acting as pointers to Off-Chain Storage) and in the Off-chain Storage (for actual data of sensitive nature or large size), using the necessary attributes per case.
DLT-SEC-17	The ABE mechanisms shall allow the decryption of data by the CONNECT components that hold the necessary attributes.

tions are ordered and consolidated into a block by the Ordering Service. This block is sent back to the Blockchain peer that checks the order and verifies the block transactions. Upon verification, the ledger is updated with a new block (Figure 4.3).

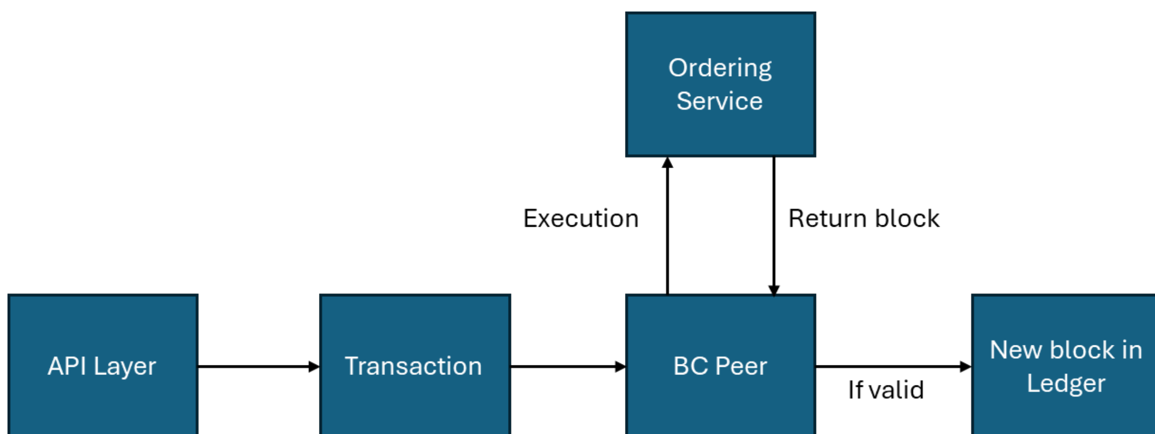

Figure 4.3: Smart contract execution

Table 4.5: Non-repudiation and Accountability requirements

SEC4 Non-repudiation and Accountability The CONNECT DLT shall enforce the accountability of all CONNECT components for their data transactions.	
Req#	CONNECT DLT SEC Requirements
DLT-SEC-18	The CONNECT DLT shall not allow the deletion or update of stored data.
DLT-SEC-19	Hashes of data transactions shall be stored in the Private Ledger towards assuring the traceability of transactions in CONNECT.
DLT-SEC-20	Data transactions in the CONNECT DLT shall be validated by a group of Blockchain Peers.
DLT-SEC-21	The consensus mechanism of the CONNECT DLT shall take into account the resolutions of the majority of Blockchain Peers.

As there are two major data flows in CONNECT DLT, the management of attestation evidence and the management of trust models/policies/RTLs, two main types of smart contracts are specified in CONNECT accordingly. In this deliverable, focus is given on the smart contracts for the attestation data flow in the next subsections, while the smart contracts for the trust policy data flow will be detailed in the next deliverable D5.3, accommodating the input from trust models and RTLs that are described in deliverable D3.2 and will be finalized in M21 of the project.

4.4.2 Data storage in CONNECT DLT (attestation evidence)

Towards showcasing a fundamental example of smart contract execution in the CONNECT DLT, the recording of failed attestation evidence coming from the AIV of a MEC (or Vehicle), is now described (see Figure 4.4). The process commences when an attestation is requested by the AIV. The actual failed attestation data are stored on the Off-chain Storage due to their rather large size in order to maintain the lightweight nature of Private Ledger. After the attestation execution and the failed result, the failed attestation data are encrypted exploiting ABE mechanisms. The ABE encryption transforms the results into a format readable solely by entities holding the suitable attributes. This ensures that even in the unlikely case that an entity manages to access the Private Ledgers and the Off-chain Storage, lacking the necessary attributes will prohibit decryption, making the attestation evidence impractical for use. Upon encrypting the attestation evidence, the AIV requests access to the CONNECT DLT to record the attestation evidence using its VCs (that have been acquired previously from CONNECT TCB). The ABAC Service determines whether the AIV is part of the CONNECT system and provided that the attributes are correct, grants access. The access permissions managed by ABAC Service are an integral part of the Security Context Broker. After being granted access, the AIV transmits the encrypted failed attestation evidence to the Private Ledger, accomplished through the API Layer, which is part of the Blockchain Peer of the CONNECT DLT. The information sent includes the attestation data, which are sent to the Private Ledger and the encrypted attestation log files for the failed result. The API Layer converts the attestation evidence into suitable output that can be digested by the Chaincode Service and the Security Context Broker. Afterwards, the hash of the attestation evidence has to be stored in the Private Ledger. The API Layer sends the hash to the Chaincode Service, which executes the transaction and conveys it to the Ordering Service. The ordered transaction block is afterwards returned to the Blockchain Peers, which commit the block in the Private Ledger. The address of the transaction block is sent back to the API Layer, completing the successful

execution of the transaction in the CONNECT DLT. After the transaction is recorded in the Private Ledger, the hash, the address of the transaction block and the encrypted attestation evidence are sent through the API Layer to the Security Context Broker, which stores the encrypted attestation evidence in the Off-Chain Storage and receives a pointer to the exact storage location of the data. This pointer is forwarded to the API Layer to execute another transaction, as the pointer is appended to the previous transaction, by performing the same flow of transaction execution and storage in the Private Ledger of the CONNECT DLT.

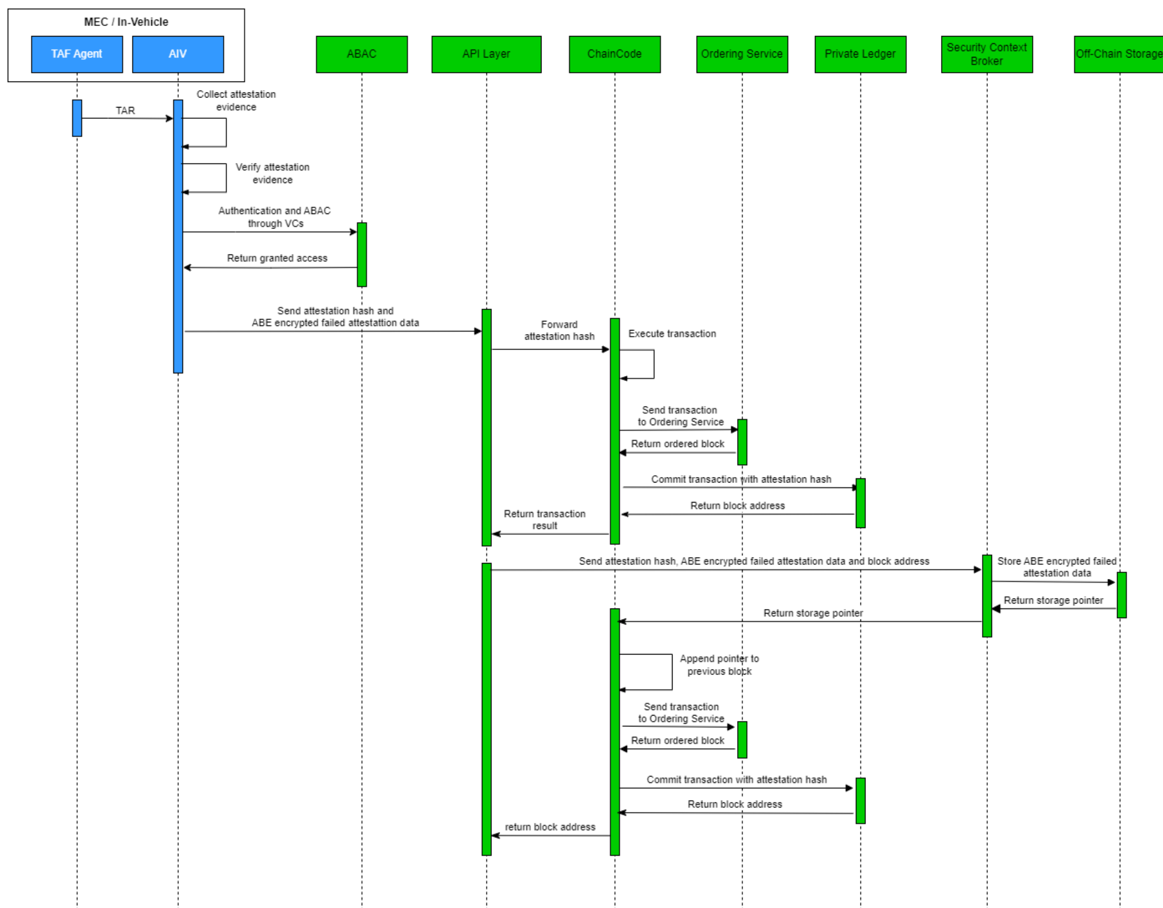

Figure 4.4: Attestation evidence recording in CONNECT DLT

4.4.3 Data retrieval in CONNECT DLT

In general, when an entity needs to retrieve information from the Private Ledger, it starts by requesting access to the CONNECT DLT through interaction with the ABAC Service. Provided the entity has the necessary VCs (Verifiable Credentials), it will be granted access and can then make a retrieval request that contains the address of the requested transaction block is forwarded to the Security Context Broker. The Security Context Broker sends this request to the Chaincode Service that transforms the query and forwards it to the Private Ledger. The Private Ledger returns the corresponding encrypted data of the specific block, which are sent back to the requesting entity through the Security Context Broker. Upon receiving the requested information, the entity can decrypt the data only if it has acquired the necessary ABE keys from the CONNECT TCB (Trusted Computing Base). In this way, it is ensured that any unauthorized entity cannot access data of the CONNECT DLT, even in the unlikely case that gets access to them. A similar process

is followed for retrieving values from the Off-Chain Storage where ABAC Service and SCB are engaged. It has to be noted that for requesting entities that are already registered in the Blockchain CA (see also section 5.1) it is not necessary to be authenticated with VCs each time they need to access the CONNECT DLT. Note that data in the Private Ledger may not be encrypted to make the DLT as lightweight as possible, hence in this case decryption is not required. However, this is not a weakness in CONNECT security as the requesting entities are authorised, and even in the unlikely case that an unauthorized component gets access to such an unencrypted information, this would be useless since unencrypted data in DLT are only pointers to encrypted data in Off-chain Storage that cannot be accessible to components without the necessary ABE keys.

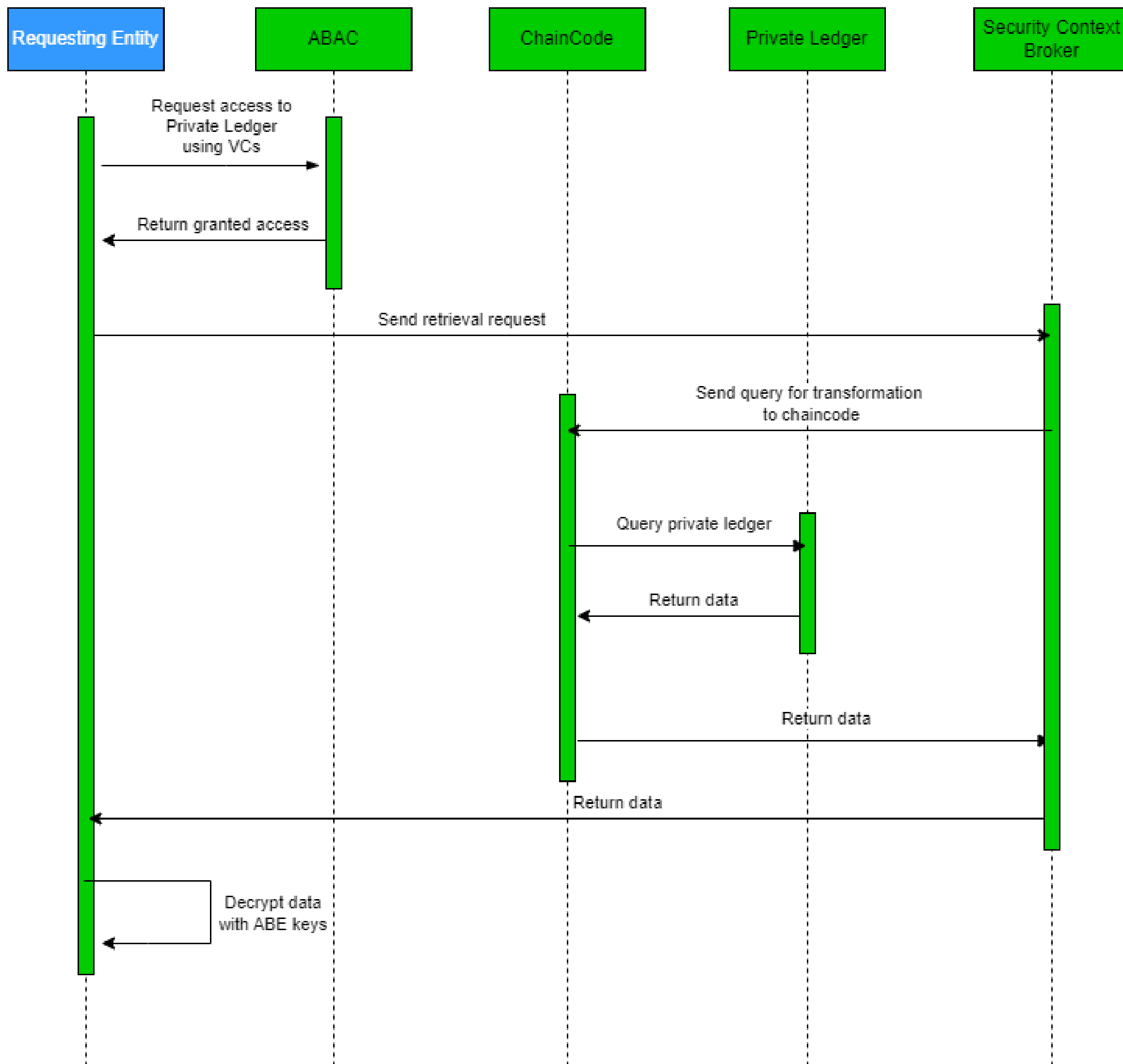

Figure 4.5: Retrieval of data from CONNECT DLT

4.4.4 Smart Contract Data Model Definition

The data models used in the context of Smart Contract definition drive the functional specifications in CONNECT DLT for the storage and retrieval of failed attestation evidence. These models are presented in the form of data structures, containing the properties that shall be included in each structure.

The specified data structures along with the fields included in each structure are described below and presented in the following tables:

Trust Policy Structure: This structure contains all required information for the definition of a Trust Policy, comprising the RTL and the trust model for the specific CCAM function of interest. After invoking the `CreateTrustPolicy()` function, the Trust Policy structure is stored in the Private Ledger (see also Table 4.6).

AttestationReport Data Structure: It is contained within the Trust Policy Structure under the name `Report`, including information regarding the failed attestation report results after the attestation is concluded, for the specific trust model included in the target policy. It is primarily used by the components which are responsible for formulating the attestation report. After invoking the `CreateAttestationReport()` function, the report is stored in the Private Ledger, under a unique PolicyID (see also Table 4.7).

AttestationPublicKeys Data Structure: This is contained within the Attestation Report Data Structure, under the name AttestationPublicKeys, including information regarding the Public Keys of the prover and the verifier that are used during an attestation process (see also Table 4.8).

Name	Data Type	JSON Schema	Description
PolicyID	`string`	‘json:”policyID”‘	String identifier used as a unique key for denoting a specific policy. Required so that devices are able to query for a specific policy, based on the required functionality to perform.
CreatedAt	`string`	‘json:”createdAt”‘	String identifier including the creation date of the policy. Used for complex queries on the DLT that are based upon creation date of a Policy.
Functions	`[]string`	‘json:”functions”‘	An array of strings that contains the functions that the policy is intended for. After being notified about the deployment of an Attestation Smart Contract, the function can query the DLT to read the Attestation Tasks of this PolicyID.
TrustModel	`[]TrustModel`	‘json:”trustModel”‘	A trust model, to be defined
ModeOfOperation	`string`	‘json:”modeOfOperation”‘	String identifier denoting the type of communication with the TAF. More specifically, it can be synchronous or asynchronous; in the former, the TAF is initiating the collection of evidence through a trust assessment request; in the latter, the TAF is notified automatically of any change in the trust state of the target device.

Table 4.6: Trust Policy Model Structure

Name	Data Type	JSON Schema	Description
ReportID	`string`	‘json:”reportID”‘	A string identifier used as a unique key for the identification of a specific attestation report. Complex queries on the DLT involving multiple reports utilize the reportID field to identify the relevant reports.
TaskID	`string`	‘json:”taskID”‘	A string identifier used to designate the TaskID of the task that was performed, and based on which the attestation report was generated.
RecordedAt	`string`	‘json:”recordedAt”‘	A string identifier including the creation date of the attestation report. Complex queries on the DLT may be based upon this field.
ListOfAccessAttributes	`[]string`	‘json:”listOfAccessAttributes”‘	This field refers to the list of attributes that a device should possess, in order to be able to read this particular attestation report.
Signature	`string`	‘json:”signature”‘	The signature contains two parts: i) The signature on the system measurements that were provided by the Prover to the Verifier, and ii) the signature by the Verifier that performs the verification and provides the results. Both these signatures are created through the Attestation Keys of their respective signed DAAs, and are used in order to verify the integrity of the data.
AttestationPublicKeys	`AttestationPKI`	‘json:”attestationPublicKeys”‘	An AttestationPKI struct, which contains x509 public key signatures of the Prover and the Verifier in string form.
TypeOfPublicKey	`typeOfPublicKey`	‘json:”typeOfPublicKey”‘	An enumeration type identifier, which is used to indicate the type of the public key, which can either be Elliptic Curve Cryptography (ECC) or RSA. Although the typeOfPublicKey is kept as an enum field (ECC or RSA) of the Attestation Report, it can be also retrieved directly from the Public Key.

Hash	string	‘json:”hash”‘	Contains the hash value of the attestation report, which is calculated by the Trustworthiness Claims Handler and signed under the newly designed anonymous threshold DAA scheme.
DBPointer	string	‘json:”DBPointer”‘	A string identifier, that is used by the SCB to store the value of the primary key of the attestation control flow after the attestation report is concluded, in the Off-chain Storage.
Metadata	[]string	‘json:”metadata”‘	An array of strings that is used by the SCB to store ABE encrypted metadata after a attestation report is stored in the DLT.

Table 4.7: Attestation Report Data Structure

Name	Data Type	JSON Schema	Description
AttestationProver	string	‘json:”attestationProver”‘	A string identifier used to hold the Public part of the Prover’s Attestation Key, which is used to sign the traced configuration or control-flow data, in order to be sent to the verifier.
AttestationVerifier	string	‘json:”attestationVerifier”‘	A string identifier that is used to hold the Public part of the Verifier’s Attestation Key, which is used to sign the attestation report, after the attestation process is complete.

Table 4.8: AttestationPublicKeys Data Structure

Chapter 5

Blockchain Data Management

5.1 Attribute-based Access Control

Towards ensuring the trustworthiness and security of information in CONNECT DLT, authentication and authorization mechanisms should be supported by robust access policy functionalities. A review on access control approaches is presented before selecting the most suitable mechanism for the CONNECT DLT. Then, a description of the selected approach is provided [28].

Attribute-Based Access Control (ABAC)

ABAC is an access control model that considers attributes of users, resources and the environment to make access decisions. It dynamically evaluates attributes against policies to determine whether a user should be granted access. ABAC provides fine-grained control over access by considering a wide range of attributes, such as user roles, location and time. Access decisions are made dynamically based on the current state of attributes, allowing for adaptable and context-aware access control. ABAC is well-suited for complex and dynamic environments where traditional access control models may be less flexible.

Role-Based Access Control (RBAC)

RBAC is a widely used access control model where access permissions are assigned based on predefined roles. Users are assigned roles and permissions associated with those roles determine what actions users are allowed to perform. RBAC simplifies access control by organizing permissions into roles, making it easier to manage user access within an organization. RBAC follows the principle of least privilege, ensuring that users have the minimum level of access necessary to perform their job functions. Access decisions are based on the roles assigned to users and these roles typically remain static until explicitly modified.

Policy-Based Access Control (PBAC)

PBAC is an access control model that uses policies to define access rules and conditions. Policies are expressed in a high-level language, allowing for expressive and flexible access control configurations. PBAC provides a high degree of flexibility by allowing the specification of access policies in a natural language or a specialized policy language. Access policies are typically managed centrally, making it easier to enforce consistent access control rules across an organization. PBAC can adapt to changes in the organization's requirements by updating or adding policies without the need for significant changes to the underlying access control infrastructure.

ABAC for CONNECT DLT

Choosing Attribute-Based Access Control (ABAC) over other access control mechanisms like Role-Based Access Control (RBAC) or Policy-Based Access Control (PBAC) in CONNECT DLT is based on the following reasons [24]:

- The granular control in ABAC allows for more precise and context-aware access decisions.
- ABAC makes access decisions dynamically based on the current state of attributes. This adaptability is valuable in CONNECT where access requirements may change frequently, enabling the system to respond to evolving conditions.
- ABAC takes into account contextual information, such as location, time of day and user attributes. This context-awareness allows for more intelligent and situation-specific access control, enhancing security and privacy of information in CONNECT DLT.
- ABAC is well-suited for scalable and adaptable access control scenarios, as it can easily incorporate new attributes and policies without the need for significant reconfiguration.
- The policies in ABAC are often expressed in terms of attributes, making them more intuitive and natural to define. This can simplify policy management and make it easier to align access control with business logic.
- ABAC can assist in meeting regulatory compliance requirements, especially in industries where access control must adhere to specific rules and conditions. The ability to define and enforce policies based on regulatory requirements is a significant advantage in CONNECT, considering the CCAM regulatory frameworks.
- ABAC can facilitate interoperability by supporting standards such as XACML (eXtensible Access Control Markup Language). This is essential in CONNECT DLT where multiple components need to operate sawith a common access control framework.

The ABAC Service

Towards entering the CONNECT DLT private channels for the first time, a new CONNECT component (e.g., a CONNECT component or a CCAM stakeholder) needs to send its attributes (issued by an external verified issuer, that can be for instance an OEM, and are encoded as VCs) to the ABAC so that the validity of the attributes issued can be verified. If this is correct, a request is forwarded to the Blockchain CA to register the CONNECT component towards receiving the correct certification. If the presented attributes have been successfully verified (i.e., been issued by a valid issuer), a new certificate is issued by the Blockchain CA and stored to the component. Upon successful verification, the CONNECT component enters the CONNECT DLT and accesses the Private Ledger. More information about the interconnection of the ABAC Service with the trust management and the leverage of verifiable credentials will be presented in the deliverable D4.3.

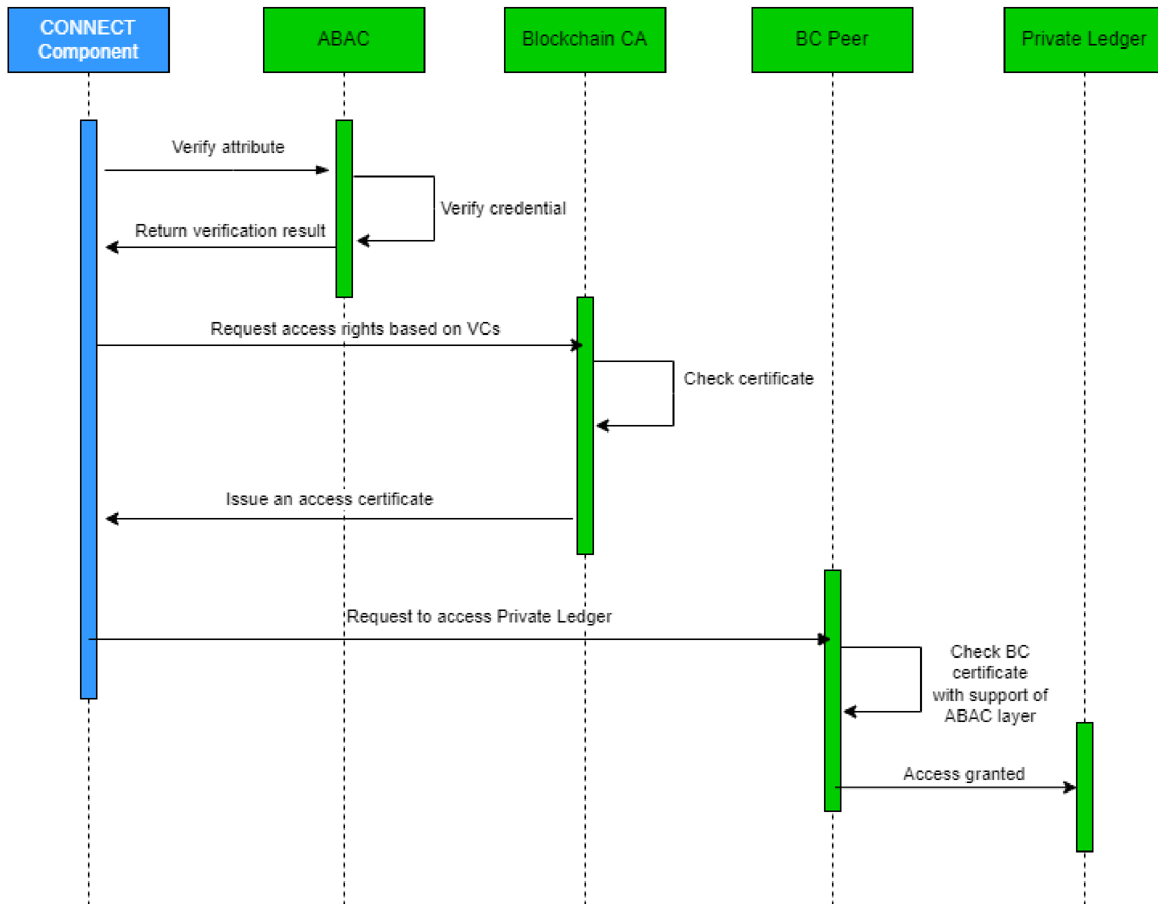

Figure 5.1: ABAC mechanism in CONNECT DLT

5.2 Attribute-based Encryption

As already mentioned in Section 4, encryption is necessary in CONNECT DLT for ensuring the security and privacy of information that is stored in CONNECT DLT. A comprehensive state-of-the-art review on Attribute-Based Encryption (ABE) schemes is provided to enable selection of the most suitable approach for CONNECT DLT, while afterwards a detailed description of the selected encryption scheme is presented.

Attribute-Based Encryption (ABE) is a cryptographic scheme that provides a flexible and fine-grained access control mechanism for encrypted data [26]. In ABE, access policies are defined using attributes and only users with the specific attributes required by the policy can decrypt the encrypted data. The attributes represent the characteristics or properties associated with users or data. The data to be encrypted is referred to as plaintext, while the encrypted form is known as ciphertext. ABE allows encryption and decryption of data based on specified attributes. Users are assigned private keys based also on their attributes. A private key is generated by the key authority and includes the user's attributes. The encryption process involves generating a ciphertext that is dependent on the access policy. Users with the corresponding attributes reflected in their private keys can decrypt the ciphertext and access the plaintext data. In ABE, data can be encrypted with respect to subsets of attributes (Key-Policy ABE) or policies defined over a set of attributes (Ciphertext-Policy ABE).

5.2.1 Ciphertext-Policy ABE

In CP-ABE, a user's private key is associated with a set of attributes and a ciphertext specifies an access policy over a defined universe of attributes within the system [9]. A ciphertext can be decrypted by a user, only in the case the policy of the corresponding ciphertext is matched by these attributes. Policies may be defined over attributes using conjunctions, disjunctions and (k, n)-threshold gates, i.e., k out of n attributes have to be present. In this sense, CP-ABE enables the definition of implicit authorization, which is contained within the encrypted data and only users who match the relevant policy can decrypt data. In addition, users can use their private keys to decrypt the data provided they satisfy one of the policies used to encrypt the data. As a result, data can be encrypted without knowing the precise users that will be able to decrypt, but only by definining the policy which enables decryption. Therefore, any users receiving a key that contain attributes satisfying the policy, will be allowed to decrypt the data.

5.2.2 Key-Policy ABE

Key-policy attribute-based encryption (KP-ABE) is similar to CP-ABE, in the sense that an access policy is encoded into secret key of the user and a ciphertext is computed based on a specific attributes [14]. An important feature that has to be addressed by KP-ABE (but also CP-ABE) is the collusion resistance. More specifically, distinct users should not be allowed to gather together their secret keys towards managing to decrypt a ciphertext together, which noone alone could decrypt. The solution to this problem is to randomize independently the secret keys of the users.

5.2.3 Decentralized ABE in CONNECT

The aforementioned two approaches are targeting at centralized ABE, where there is a single central entity that manages the key generation, attribute policies and access control decisions. This central entity is responsible for generating and distributing secret keys based on users' attributes. However, centralized ABE approaches may face scalability challenges as the central authority becomes a potential bottleneck. Also, as trust is concentrated in this central entity, its compromise could have significant security implications (single point of failure).

As such, decentralized ABE comes as a necessity in CONNECT, where trust management and system reputation are at the forefront. The decentralized ABE involves multiple distributed entities that collaboratively contribute to the key generation, policy enforcement and access control decisions. Unlike centralized ABE, trust is distributed across multiple entities. This can enhance the security of the system, as the compromise of a single authority may not lead to a complete breach. Decentralized ABE can provide better scalability, as the responsibilities are distributed. It also provides redundancy, making the system more resilient to failures or attacks on individual entities. Also, access policies may involve a collaborative effort among multiple authorities, each contributing a part of the policy [18].

In this context, a high-level conceptual overview of the decentralized ABE scheme for CONNECT DLT is presented below. Notably, the scheme consists of the three following phases, each of them including various steps:

Initialization

(A) A device in the ABE scheme may be either an encryptor or a decryptor. The device commences by sending its static attributes to be used as part of verifiable presentations to the Security Context Broker.

(B) A key is created by the SCB for the device to be put into its key hierarchy along with the static attribute elliptic keys towards defining the encryption/decryption and HMAC keys under a key policy, including a concatenation of dynamic and static attributes.

(C) This data is encrypted under the public part of the endorsement key of the device.

(D) When the encrypted data is obtained by the device, it decrypts and stores the attribute keys, while putting the HMAC key provided by the SCB in its key hierarchy.

Encryption

(A) To encrypt data with ABE, an encryptor creates a random nonce.

(B) This nonce is used to calculate along with the master attribute key and the key restriction policy, the encryption key seed.

(C) The encryption key is created under the key provided by the SCB, bound with the appropriate policy.

(D) The device undergoes an integrity check, through the local attestation capabilities offered by the underlying CONNECT TCB (see deliverable D4.2), towards encrypting the data, using its list of attributes.

(E) Upon succeeding in the integrity check, the device encrypts the data and creates its HMAC.

(F) The device calculates a shared value, using the public static attribute keys, and the previously created random nonce.

(G) To perform a DAA signature operation on the key restriction policy, the device uses its DAA key towards offering a security proof that the correct key restriction policy was employed during the encryption of the raw traces.

(H) The device sends the encrypted data, the shared value, the HMAC, the policy signature, the public part of the signing key and its corresponding credential to the SCB to be stored in the Off-chain Storage.

(I) The policy signature used for the encryption along with the public part of the signing key, are stored in the Private Ledger.

Decryption

(A) The decryptor retrieves the DAA signature in order to decrypt a set of ABE encrypted data.

(B) The device uses the policy expected to be used for the verification of the signature provided by the encryptor and the recreation of the credential.

(C) Upon successful verification, the decryptor calculates the encryption key seed using its static attribute key.

(D) The decryptor creates an HMAC key using the extracted key seed to recalculate the HMAC of the encrypted data.

(E) The decryptor runs an authentication check by comparing this HMAC with the one provided by the encryptor.

(F) Upon successful authentication, an integrity check of the decryptor is performed with the dynamic and static attributes.

(G) Since the integrity check is successful, the device decrypts the data with the extracted attribute keys.

5.2.4 Overview of the CONNECT ABE scheme

Towards describing the CONNECT ABE scheme, the notation that is used within this section is presented in Table 5.1.

Notation	Description
$KhEk_A$	Handle of the Endorsement Key of device A
Ek_{pub}	Public part of the Endorsement Key
$KhEk$	Handle of the endorsement key of the SCB (authority)
$KHash$	Keyed hash key
$KHash_{pub}$	Public part of the keyed hash key
$KhHash$	Handle of the keyed hash key
SH	Session handle
DCC	Command code of tpm2_duplicate
$randP$	Random 32 bytes password
$Priv$	The encrypted, by the random password and a TPM symmetric key, private part of the keyed hash key
$symSeed$	The encrypted (with the device's endorsement key) seed that created the symmetric key that encrypted the private part of the keyed hash
$sAbT$	Static attribute based on tree
rtA	Run-time attributes
Cre	The credentials needed for the initialization of the device
P	The policy that includes both the static attributes and the run-time attributes
K	A random nonce created by the TPM
$Secret$	The seed for the creation of the encryption and the HMAC key
PK	The public master attribute key
$KhEnc$	The handle of the symmetric encryption key
$HMAC_{key}$	The handle of the HMAC key
$HMAC$	The HMAC of the encrypted data
$HMACp$	The recreated HMAC of the encrypted data
Sig	The signature of the session digest
CT	CipherText
t_i	An attribute of the static attribute-based tree
P_i	The public attribute key of t_i
$shared$	The shared value that contains the static attributes that were used
D	The set of leaves of the static attribute based tree that will be used for the extraction of the secret
R	The root of the static attribute based tree
PT	Plaintext
SH	Session handle
$Nonce$	Session nonce
$SigPub$	Public part of the SCB's signing key

nP	New policy
$CpHash$	$H(nonce\|nP)$
$KhSigK$	Sey handle of the signing key of the SCB
$SigP$	Signature of the new policy

Table 5.1: Notations used in the description of the CONNECT ABE scheme.

An overview of the ABE scheme employed in CONNECT DLT (Figure 5.2) is described in the following subsections, leveraging on the details of each different phase. More information will be provided regarding the ABE implementation for the CONNECT DLT in the deliverable D5.3.

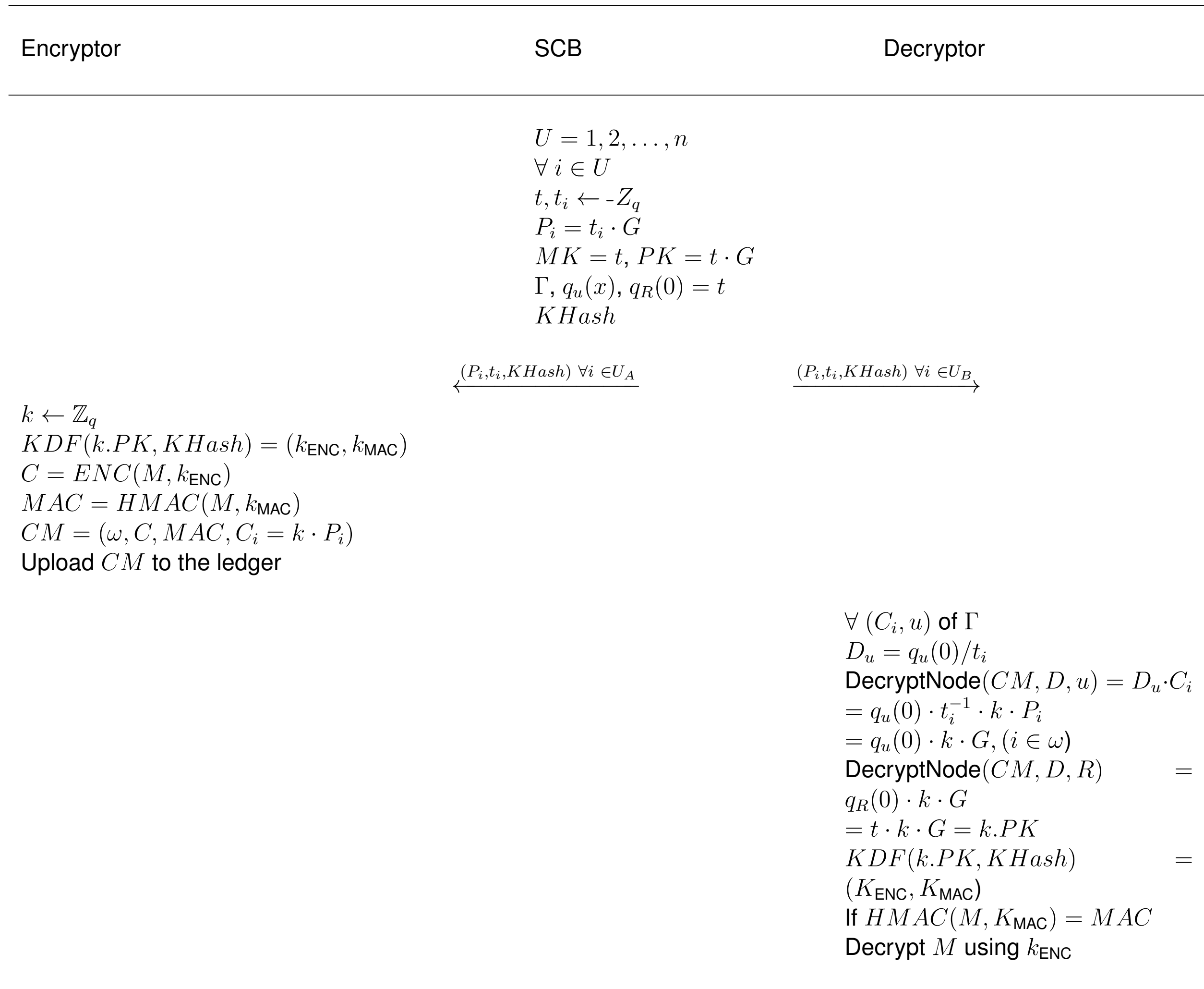

Figure 5.2: Overview of the CONNECT ABE scheme

5.2.4.1 Initialization

The setup interface is commenced by the Security Context Broker (SCB) and is performed when a device (encryptor or decryptor) joins the ABE scheme. As the trusted authority, the SCB verifies

the attributes by their verifiable presentations during the secure enrollment of the device, generating the respective static attribute keys for each CONNECT component (device), as well as defining the access control tree. An example of the access control tree is presented in Figure 5.3 (t and t_i, $i \in U$, are represented by 32-byte digests).

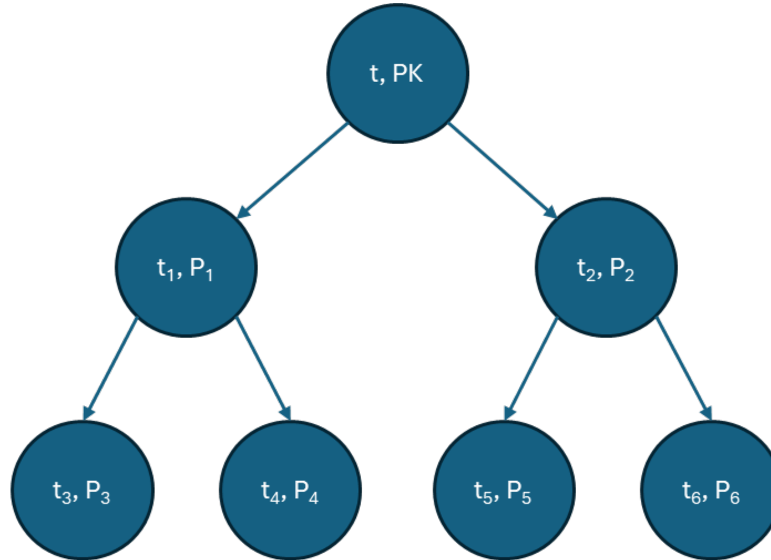

Figure 5.3: Access control tree created by the SCB

The sequence of steps of the Initialization phase are described below:

(A) For the setup of the SCB, a keyed hash key is created as a child of the endorsement key of the SCB. This key is migrated to every device that is securely enrolled to the SCB, to be used as the parent of the encryption and the HMAC keys under the key hierarchy of the device.

(B) The attribute space in the system is defined as the universe of attributes $U = 1, 2, ..., n$. For each attribute $i \in U$, a number t_i is chosen uniformly at random from $\mathbb{Z}_q$. The public key of each attribute i is $P_i = t_i \cdot G$, where G is the group generator. The group generator is public information that is known to all devices.

(C) The SCB chooses t uniformly at random from $\mathbb{Z}_q$ to be the master private key t. Accordingly, the master public key PK is $PK = t \cdot G$. The public parameters are represented by $Params = PK, P_1, ..., P_U$.

(D) The SCB defines an access tree Γ by creating a public polynomial $q_u(x)$ with order of $d_u - 1$ for each node u in the access tree Γ in top-down manner such that $q_u(0) = q_{parent}(index(u))$, where d_u is the threshold of the node u. For the root R of the access tree Γ, set $q_R(0) = t$.

In order to prevent the CPA attack presented in [29], the creation of q_u at each node, should not only depend on the index of the node u in the access tree, but also on an extra n-bit randomness $r_{\mathbb{A}}$ that depends on the access policy defined at this node. Thus, q_u at the node u is defined by $q_u(PRF(r_{\mathbb{A}}, index_u))$ for some pseudo-random function $PRF : \{0,1\}^n \times \{0,1\}^* \rightarrow \mathbb{Z}_q^*$ that takes a random $r_{\mathbb{A}}$ and the index of the node where the polynomial is defined.

(E) For the setup of the device (e.g., a vehicle ECU), the device's DAA Key is used, which is created during the secure enrollment for the encryption of the data that is not public and that the SCB securely sends to the device.

(F) The device sends to the SCB the public part of its endorsement key along with the name of the DAA Key.

(G) The SCB generates a random password, used to double encrypt the private part of the keyed hash, comprising an extra level of security for freshness every time a device challenges the SCB to duplicate the keyed hash key. The SCB defines a random seed that afterwards creates a symmetric key, which encrypts the private part of the key. The random seed is encrypted with the public part of the endorsement key of the device that initiated the challenge. Finally, the encrypted private part of the keyed hash is encrypted with the random password generated by the SCB.

(H) The SCB encrypts the static attribute-based keys and the random password for each user under the endorsement key of the device and wraps them with the name of the DAA Key.

(I) The SCB generates a Key Policy, which is the concatenation of the attributes defined by the corresponding OEM of ECU, along with the command codes of the commands that need to be executed. This Key Policy is bound with the encryption/decryption key to provide a check on the integrity of the device.

(J) The SCB sends the respective Key Policy to the device, containing the list of attributes as well as the encrypted attribute-based access control tree, in order for the device to be able to calculate the random key seed for the encryption key and the HMAC key and to compute the shared value from which the decryptor extracts the random key seed, along with the random password, and the double encrypted private part of the keyed hash and the encrypted (by the public part of the endorsement key of the device) seed that is used to generate the inner password and the public part of the keyed hash.

(K) The device decrypts the encrypted static attribute-based tree and the random password generated by the SCB.

(L) The device using the decrypted random password, the public part of the keyed hash, the encrypt seed of the inner encryption key and the double encrypted private part of the keyed hash safely imports the SCB keyed hash as a child of its endorsement key.The keyed hash is imported to every device in order to support the creation of derived keys.

5.2.4.2 Encryption

During the Encryption phase, the encryptor encrypts data to be uploaded on the CONNECT DLT, following the steps below:

(A) The encryptor creates a random secret k and performs the scalar multiplication $k * PK$.

(B) The encryptor takes the secret value $k * PK$, which is used as a co-seed along with the private part of the imported keyed hash key, to create a symmetric encryption key bonded with the key restriction policy that the SCB has created during the initialization phase and an HMAC key, as children of the Keyed hash key.

(C) The device loads the static attributes to a session. It also uses the run-time attributes to check the key restriction policy, that the SCB has created during the initialization of the protocol, against the policy that is created from the static attributes and run-time attributes taken as input.

(D) The device encrypts the data with the newly created encryption key.

(E) The encryptor uses the DAA key to perform a DAA signature for the audited session digest, towards using this signature as a proof of the correct policy usage.

(F) The device computes the digest of the encrypted data With the newly created HMAC key.

(G) The encryptor reads the static attributes and computes the shared value $C_i = kP_i$ for $i \in \omega$, where $\omega \subset U$ is a set of required attributes for the encryptor.

(H) The encryptor uploads the encrypted data, the HMAC of the encrypted data and the shared value C_i.

5.2.4.3 Decryption

A device can decrypt data uploaded by an encryptor on the CONNECT DLT using matching attributes, following the steps below:

(A) The decryptor that has the correct policy that had to be used during the encryption of the data, verifies the DAA signature that the encryptor provided. Upon successful signature verification, it is proven that the encryptor encrypted the data under the correct policy, thus the decryptor can continue the decryption algorithm.

(B) For each C_i on the nodes u of Γ, the decryptor computes the corresponding decryption node key denoted by D: $D_u = q_u(0)/t_i$. Notably, $i = attr(u)$ and t_i is the attribute key assigned to users from $\mathbb{Z}_q$ in the Setup phase by the SCB, and t_i^{-1} is the inverse element of t_i over finite field $\mathbb{Z}_q$.

(C) The decryptor executes the decryption algorithm $DecryptNode(C_i, D, u)$ for a node u in the access tree, which is defined as an recursive procedure. For a leaf node u, where attributes are clearly defined, let $i = attr(u)$. an For this node, the key is computed as $DecryptNode(C_i, D, u) = D_u * C_i = q_u(0) \cdot t_i^{-1} \cdot k \cdot P_i$. It should be stated that the output of $DecryptNode(C_i, D, u)$ is an element in an elliptic curve group or $\perp$. Accordingly, for the root node R of the access tree, the key is computed as $DecryptNode(C_i, D, R) = q_R(0) \cdot k \cdot G = t \cdot k \cdot G = k \cdot PK$.

(D) The decryptor, with the extracted secret value $k * PK$, uses $k * PK$ as a co-seed, along with the private part of the imported from the SCB keyed hash, to create the decryption key bound by the policy provided by the SCB and the HMAC Key, as children of the keyed hash key.

(E) The decryptor computes the digest of the encrypted data, using the HMAC key, and compares it to the one provided by the encryptor.

(F) Upon successful authentication, the decryptor loads the list of atrributes needed for getting access to this specific data item.

(G) Upon success of the integrity check, the decryptor decrypts the encrypted message using the recreated decryption key.

Chapter 6

Conclusions

The document delivers the first version of the CONNECT orchestration technology. It elaborates on the combination of two CONNECT artifacts a) the distributed management of (legacy) application containers and b) the confidential containers lifecycle management techniques. Their interplay has been the main objective of the CONNECT networking technology implementation. The above will act as the main vehicle towards realizing all involved communications among the CONNECT points of interest (i.e., cloud, MEC and vehicles) and also support the task-offloading concept; the latter has been also analysed in terms of a generic algorithm as well as practical considerations to comply with the CONNECT demonstrator requirements/needs.

A second delivered body of work in this document amounts to the design of the distributed ledger technology that facilitates the CONNECT trusted and secure data sharing; both for attestation results and trust-related data being hosted in the blockchain infrastructure. These allow CONNECT to achieve auditability (through a verifiable yet privacy-preserving manner). All relevant details as well as the appropriate access policies to allow for the handling/monitoring of those (current or historical) data (e.g., past trustworthiness evidence), have been specified in this deliverable, serving as a basis for their forthcoming implementation (to be detailed in D5.3).

When all above delivered contributions are combined with the utilisation of Verifiable Credentials and Verifiable Presentations (D5.1) which rely on an efficient data model to capture all involved evidence, the WP5 manages to already reach a highly-matured (i.e., towards integration-ready) outcome; both the CONNECT-required communications can be supported by managing confidential containers, the ledger-based storage of important CONNECT data has been specified and at the same time the necessary tools (D5.1) to drive the (runtime) trust assessment process have been offered. The finalisation (second version) of the CONNECT orchestrator and the ledger software is expected to build on top of the aforementioned contributions and constitute one important means towards the realisation of the CONNECT integration and demonstrator (WP6).

Chapter 7

List of Abbreviations

Abbreviation	Translation
5G	Fifth-generation Cellular Networks Technology
ABAC	Attribute-Based Access Control
ABE	Attribute-Based Encryption
AIV	Attestation and Integrity Verification
API	Application Programming Interface
ATL	Actual Trust Level
BC	Blockchain
CA	Certification Authority
CCAM	Connected, Cooperative, and Automated Mobility
CRI-O	Container Runtime Interface (implementation)
CPA	Chosen-Plaintext Attack
CP-ABE	Ciphertext-Policy ABE
CPU	Central Processing Unit
C-ACC	Cooperative Adaptive Cruise Control
DAA	Direct Anonymous Attestation
DLT	Distributed Ledger Technology
ECC	Elliptic Curve Cryptography
EVM	Ethereum Virtual Machine
GPU	Graphics Processing Unit
HLF	Hyperledger Fabric
HMAC	Hash-Based Message Authentication Codes
HW	Hardware
IAM	Identity and Authentication Management
ICT	Information and Communications Technology
IP	Internet Protocol
KPI	Key Performarce Indcator

Abbreviation	Translation
K8s	Kubernetes
LCM	Lifecycle Management
MANO	Management and Orchestration
MEC	Multi-Access Edge Computing
MECsec	Media Access Control Security
ML	Machine Learning
MNO	Mobile Network Operator
OCI	Open Container Interface
OEM	Original Equipment Manufacturer
OS	Operating System
PBAC	Policy-Based Access Control
PBFT	Practical Byzantine Fault Tolerance
PKI	Public Key Infrastructure
PoA	Proof of Authority
PoW	Proof of Work
RA	Risk Assessement
RAM	Random-Access Memory
RBAC	Role-Based Access Control
RO	Resoure Orchestrator
RSA	Rivest–Shamir–Adleman cryptosystem
RTL	Required Trust Level
RTT	Round Trip Time
SCB	Security Context Broker
SMTD	Slow Moving Traffic Detection
SW	Software
TAM	Trust Assessment Manager
TAF	Trust Assessment Framework
TC	Trustworthiness Claim
TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
VC	Verifiable Credentials
VP	Verifiable Presentation

Bibliography

[1] UN Regulation No. 155. Cyber security and cyber security management system. https://unece.org/transport/documents/2021/03/standards/un-regulation-no-155-cyber-security-and-cyber-security.

[2] ISO/SAE 21434:2021. Road vehicles Cybersecurity engineering. https://www.iso.org/standard/70918.html.

[3] 5GAA. C-V2X use cases and service level requirements (2023). https://5gaa.org/c-v2x-use-cases-and-service-level-requirements-volume-iii/.

[4] Abdelkader Aissioui, Adlen Ksentini, Abdelhak Mourad Gueroui, and Tarik Taleb. On enabling 5G automotive systems using follow me Edge-Cloud concept. *IEEE Transactions on Vehicular Technology*, 67(6):5302–5316, 2018.

[5] Oğuzhan Akyıldız, Feyza Yıldırım Okay, İbrahim Kök, and Suat Özdemir. Road to efficiency: Mobility-driven joint task offloading and resource utilization protocol for connected vehicle networks. *Future Generation Computer Systems*, 2024.

[6] Artem Barger Vita Bortnikov Christian Cachin Konstantinos Christidis Angelo De Caro David Enyeart et al Androulaki, Elli. Hyperledger fabric: a distributed operating system for permissioned blockchains. *In Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[7] Mate Boban, Apostolos Kousaridas, Konstantinos Manolakis, Josef Eichinger, and Wen Xu. Connected roads of the future: Use cases, requirements, and design considerations for vehicle-to-everything communications. *IEEE Vehicular Technology Magazine*, 13(3):110–123, 2018.

[8] Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *ZEUS*, pages 65–73, 2021.

[9] Li J. Zhang Y. Guo Y. Chen, N. Efficient cp-abe scheme with shared decryption in cloud storage. *IEEE Transactions on Computers*, 71(1):175–184, 2020.

[10] The CONNECT Consortium. D2.1: Operational Landscape, Requirements and Reference Architecture – Initial Version. 2023.

[11] Lin C. Khazaei H. Musilek P. Fan, C. Performance analysis of hyperledger besu in private blockchain. *IEEE international conference on decentralized applications and infrastructures (DAPPS)*, pages 64–73, 2022.

[12] Stephen L Jones, Stanley E Fawcett, Amydee M Fawcett, and Cynthia Wallin. Benchmarking trust signals in supply chain alliances: moving toward a robust measure of trust. *Benchmarking: An International Journal*, 17(5):705–727, 2010.

[13] Kumar N. Kaushal, R. K. Blockchain implementation with hyperledger fabric and approach for performance evaluation. *IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS)*, pages 1–5, 2023.

[14] Susilo W. Guo F. Au M. H. Nepal S. Kim, J. An efficient kp-abe with short ciphertexts in prime ordergroups under standard assumption. *In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 823–834, 2017.

[15] Vojdan Kjorveziroski and Sonja Filiposka. Kubernetes distributions for the edge: serverless performance evaluation. *J. Supercomput.*, 78(11):13728–13755, jul 2022.

[16] Heiko Koziolek and Nafise Eskandani. Lightweight kubernetes distributions: A performance comparison of microk8s, k3s, k0s, and microshift. In Marco Vieira, Valeria Cardellini, Antinisca Di Marco, and Petr Tuma 0001, editors, *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE 2023, Coimbra, Portugal, April 15-19, 2023*, pages 17–29. ACM, 2023.

[17] Tra Huong Thi Le, Nguyen H. Tran, Tuan LeAnh, Thant Zin Oo, Kitae Kim, Shaolei Ren, and Choong Seon Hong. Auction mechanism for dynamic bandwidth allocation in multi-tenant edge computing. *IEEE Transactions on Vehicular Technology*, 69(12):15162–15176, 2020.

[18] Waters B. Lewko, A. Decentralizing attribute-based encryption. *In Annual international conference on the theory and applications of cryptographic techniques*, pages 568–588, 2011.

[19] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2021.

[20] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys Tutorials*, 19(3):1628–1656, 2017.

[21] Aaron Miller, Kyungzun Rim, Parth Chopra, Paritosh Kelkar, and Maxim Likhachev. Cooperative perception and localization for cooperative driving. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1256–1262, 2020.

[22] Sanghoon Oh, Linjun Zhang, Eric Tseng, Wayne Williams, Helen Kourous, and Gabor Orosz. Safe decision and control of connected automated vehicles for an unprotected left turn. In *Dynamic Systems and Control Conference*, volume 84270. American Society of Mechanical Engineers, 2020.

[23] Hassan A. E. Jiang Z. M. Oliva, G. A. An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, 25:1864–1904, 2020.

[24] M. Penelova. Access control models. *Cybernetics and Information Technologies*, 21(4):77–104, 2021.

[25] Marfievici R. McGibney A. Rea S. Ranathunga, T. A dlt-based trust framework for iot ecosystems. *International Conference on Cyber Security and Protection of Digital Services (Cyber Security) IEEE*, pages 1–8, 2020.

[26] La Manna M. Perazzo P. Dini G. Rasori, M. A survey on attribute-based encryption schemes suitable for the internet of things. *IEEE Internet of Things Journal*, 9(11):8269–8290, 2022.

[27] Git Repository. Process-based confidential container runtime. https://github.com/confidential-containers/enclave-cc.

[28] Osborn S. L. Servos, D. Current research and open problems in attribute-based access control. *ACM Computing Surveys (CSUR)*, 49(4):1–45, 2017.

[29] Syh-Yuan Tan, Kin-Woon Yeow, and Seong Oun Hwang. Enhancement of a lightweight attribute-based encryption scheme for the internet of things. *IEEE Internet of Things Journal*, 6(4):6384–6395, 2019.

[30] Teltonika. Industrial 5G Router. https://teltonika-networks.com/products/routers/rutx50.

[31] Car to Car Communication Consoritum. Day 2 use cases and beyond (2023). https://www.car-2-car.org/fileadmin/documents/General_Documents/C2CCC_UC_2097_UseCases_V1.0.pdf.

[32] James Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018.

[33] J.J. Verbakel, M. Fusco, D.M.C. Willemsen, J.M. van de Mortel-Fronczak, and W.P.M.H. Heemels. Decision making for autonomous vehicles: Combining safety and optimality. *IFAC-PapersOnLine*, 53(2):15380–15387, 2020. 21st IFAC World Congress.

[34] Zhiwei Zhang, Zehan Chen, Yulong Shen, Xuewen Dong, and Ning Xi. A dynamic task offloading scheme based on location forecasting for mobile intelligent vehicles. *IEEE Transactions on Vehicular Technology*, pages 1–15, 2024.