SOFIE - Secure Open Federation for Internet Everywhere

779984

DELIVERABLE D2.7

Federation Framework, final version

Project title	SOFIE – Secure Open Federation for Internet Everywhere
Contract Number	H2020-IOT-2017-3 – 779984
Duration	1.1.2018 – 31.12.2020
Date of preparation	12.5.2021
Author(s)	Yki Kortesniemi, Dmitrij Lagutin & Wu Lei (AALTO), Nikos Fotiou, Iakovos Pittaras, Spyros Voulgrais & Vasilios A. Siris (AUEB-RC), Giuseppe Raveduto (ENG), Mait Märdin (GT), Antonio Antonino & Filippo Vimini (LMF), Ahsan Manzoor (ROVIO), Yannis Oikonomidis (SYN)
Responsible person	Yki Kortesniemi (AALTO), Yki.Kortesniemi@aalto.fi
Target Dissemination Level	Public
Status of the Document	Completed
Version	1.10
Project web-site	https://www.sofie-iot.eu/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779984.

Summary of changes

Version	Major changes
1.10	The major changes from version 1.00 include: 1) Added clarifications to sections 1, 2 and 4 that as a framework/meta architecture, the SOFIE Architecture only defines the functionalities each component should provide, but no APIs or implementation for the functionality - each system is free to use the most suitable technologies for their implementation. The SOFIE Framework provides one example implementation for each component. 2) The requirements tables now also contain information about which pilot was the source for the requirement. 3) Added information on the validation of Architecture requirements also through the validation of pilots in D5.4 4) Added details to Section 2.2 about the development of the Framework and how the TRL levels of the components were evaluated 5) Added details about the scalability of the Interledger component in Section 3. 6) Added details to Section 4 about how the permissions are managed and about the test cases. 7) Expanded the description of WoT TDs and their use in multi-domain situations in Section 6.

Table of Contents

List of abbreviations

API	Application Programming Interface
BLE	Bluetooth Low Energy
DID	Decentralised Identifier
DLT	Distributed Ledger Technology
DNS	Domain Name Server
DSO	Distribution System Operator
EV	Electrical vehicle
FSC	Food Supply Chain
HTLC	Hash Time-Lock Contract
IAA	Identity, Authentication, authorisation [component]
IL	Interledger [component]
IoT	Internet of Things
KSI	Keyless Signing Infrastructure (GuardTime)
MP	Marketplace [component]
P&D	Provisioning and Discovery [component]
PDS	Privacy and Data Sovereignty [component]
RFID	Radio Frequency IDentification
SR	Semantic Representation [component]
TD	Things Descriptor
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
WLAN	Wireless Local Area Network
WoT	Web of Things

1 Introduction

SOFIE (Secure Open Federation for Internet Everywhere) is a three-year EU Horizon 2020 research and innovation project that provides interoperability between existing IoT systems in an open and secure manner.

The *SOFIE Architecture* described in the SOFIE Deliverable '2.6 - Federation Architecture, final version' [D2.6] achieves interoperability by *federating the actions between different IoT systems using interledger technologies*. The Architecture consists of 1) 6 components that enable key functionalities for federation scenarios and 2) federation adapters used to connect the IoT systems to the architecture *without requiring changes to the IoT devices*. The architecture can be extended to support different use cases and the individual components can be implemented using technologies that best suit the context.

This document, which supersedes the earlier SOFIE Framework deliverable D2.5 [D2.5], introduces the *SOFIE Federation Framework,* an example implementation of the Architecture available as open-source software in GitHub [Framework]. It describes all the components'/adapters' purpose, interfaces, and internal structure, how the SOFIE pilots leverage the components/adapters, and how the components have been validated against the requirements set in D2.6.

The rest of the document is organised as follows: Section 2 presents an overview of the SOFIE Architecture and Framework. Then, Sections 3-9 detail the 6 components and the federation adapters and their validation. Finally, Section 10 describes how the SOFIE pilots utilise these components and Section 11 summarises the document.

2 SOFIE Architecture

This section provides an overview of the SOFIE Architecture, summarises the requirements for the Architecture, and reports the validation test results that show the requirements have all been met.

2.1 Requirements and Validation

Table 2.1 details the functional requirements and Table 2.2 details the privacy requirement for the SOFIE Architecture (from SOFIE deliverable D2.6).The sources for the requirements have been indicated in the tables as follows: SOFIE Description of Action (DoA), legislation (L), project pilots (FSC/DEFM/CAMG/DEDE) and the reference application SMAUG (S).

Table 2.1 Requirements for the SOFIE Architecture

Req. ID	Requirement Description	Priority	Category	Source
RA01	SOFIE Architecture must define a clear separation between data management, control, and representation processes.	MUST	QUALITY	FSC, DEFM, CAMG, DEDE
RA02	SOFIE Architecture must be modular to enable different use cases and reuse of components.	MUST	QUALITY	FSC, DEFM, CAMG, DEDE, S
RA03	The interfaces of the SOFIE components must be well-defined and fully documented.	MUST	QUALITY	FSC, DEFM, CAMG, DEDE, S
RA04	Transactions must be immutable and verifiable. Parties must not be able to modify existing transactions without other parties noticing it. Every party should be able to independently verify the validity of transactions.	MUST	SECURITY	FSC, DEFM, CAMG, DEDE, S
RA05	The system must provide auditability.	MUST	SECURITY	FSC, DEFM, CAMG, DEDE, S
RA06	Support for transactions, where only authorised entities can participate. Minimal amount of information should be disclosed during authentication.	MUST	SECURITY	FSC, DEFM, DEDE, S
RA07	All external and internal interfaces and communication links of the system must conform to the principle of least privilege.	MUST	SECURITY	FSC, CAMG, S
RA08	The SOFIE Architecture should be flexible and support different means of user authentication, including password-based, certification-based, and token-based.	SHOULD	SECURITY	FSC, DEDE, S

Table 2.2. Privacy requirements for implementation and deployment of the SOFIE Architecture

Req. ID	Requirement Description	Priority	Category	Source
RP01	Privacy issues and business secrets must be considered carefully when deciding what data (including authentication/ authorisation information, logs etc.) is collected, stored or exchanged between parties.	MUST	POLICY & REGULATION	L, FSC

The validation showing that the Architecture meets all the functional requirements have been summarised in Table 2.3. Due to their qualitative nature, many of the requirements have been validated based on documentation, but the results are further validated by the successful validation of the pilots utilising the Architecture as detailed in D5.4 [D5.4].

Table 2.3 SOFIE architecture validation matrix

ID	***Validation Process***	
RA01	*Requirement Description*	SOFIE architecture must define a clear separation between data management, control, and representation processes.
	Test approach	Documentation
	Test Description	Architecture is divided into multiple components that carry out the different processes in separation of each other.
	Test location	D2.6, Section 2.2, page 11
RA02	*Requirement Description*	SOFIE architecture must be modular to enable different use cases and reuse of components.
	Test approach	Documentation
	Test Description	Architecture is divided into extendable components. SOFIE pilots are examples of how the components can be used for different use cases.
	Test location	D2.6, Section 2.2, page 11 & Section 3, page 13-
RA03	*Requirement Description*	The interfaces of the SOFIE components must be well-defined and fully documented.
	Test approach	Documentation
	Test Description	Component interfaces are described in the Framework documentation.
	Test location	This document, Sections 3-8, page 11-
RA04	*Requirement Description*	Transactions must be immutable and verifiable. Parties must not be able to modify existing transactions without other parties noticing it. Every party should be able to independently verify the validity of transactions.
	Test approach	Functional test

	Test Description	Event on one ledger automatically triggers the transfer of data/asset to another ledger. All information (initiation of transfer, acceptance of transfer, confirmation of acceptance) is stored on the ledgers, which makes the information immutable and verifiable.
	Test location	https://github.com/SOFIE-project/Interledger/blob/master/tests/system/test_interledger_ethereum.py
RA05	*Requirement Description*	The system must provide auditability.
	Test approach	Functional test
	Test Description	The SMAUG reference application implements auditability for all the key functionalities.
	Test location	https://github.com/SOFIE-project/SMAUG-Deployment/blob/master/docs/pages/requirements_validation.md
RA06	*Requirement Description*	Support for transactions, where only authorised entities can participate. Minimal amount of information should be disclosed during authentication.
	Test approach	Documentation
	Test Description	Architecture (through the IAA component) can be configured to support any type of authorisation server including servers supporting minimal disclosure of information.
	Test location	D2.6, Section 4.2, page 21
RA07	*Requirement Description*	All external and internal interfaces and communication links of the system must conform to the principle of least privilege.
	Test approach	Documentation
	Test Description	Architecture has been designed with the principle of least privilege.
	Test location	D2.6, Section 4.3, page 22
RA08	*Requirement Description*	The SOFIE architecture should be flexible and support different means of user authentication, including password-based, certification-based, and token-based.
	Test approach	Documentation
	Test Description	IAA component can be configured to support any type of authorisation server including servers supporting different means of authentication.
	Test location	D2.6, Section 4.2, page 21

Finally, although satisfying the privacy requirement is ultimately up to each implementation and deployment, the SOFIE Architecture *provides all the necessary tools* to this end. For instance, using the Interledger component it is possible to store all confidential information only in each organisation's private storage and then store a hash to a public ledger for increased trust. If a dispute requires auditing the information in the private storage, the hash proves that the data was not modified after the hash was published. Further, the Privacy and

Data Sovereignty component also supports privacy preserving surveys using differential privacy, thus avoiding the storage of privacy compromising information in the first place.

2.2 Architecture overview

The lack of interoperability between IoT systems has long been a significant limitation to the creation of solutions spanning multiple IoT systems. This situation has been further aggravated by the fact that adding interoperability to existing IoT systems can be hard due to some systems not being upgradable. The *SOFIE Architecture* enables interoperability between existing IoT systems in an open and secure manner by *federating the actions between different IoT systems using interledger technologies*. The Architecture consists of 6 components that enable key functionalities for federation scenarios and of federation adapters used to connect the IoT systems to the architecture *without requiring changes to the IoT platforms and devices*. The architecture can be extended to support different use cases and the individual components can be implemented using technologies that best suit the context. The Architecture has been detailed in SOFIE Deliverable D2.6 [D2.6].

The *SOFIE Framework* detailed in this document is an example implementation of the Architecture designed to support the SOFIE pilots (the pilots are described in more detail in the SOFIE Deliverable D5.4 [D5.4]) and to fulfill the requirements set in D2.6. The full Framework, detailed technical documentation, and multiple examples of how the components and adapters can be utilised are all available as open-source software in the GitHub [Framework].

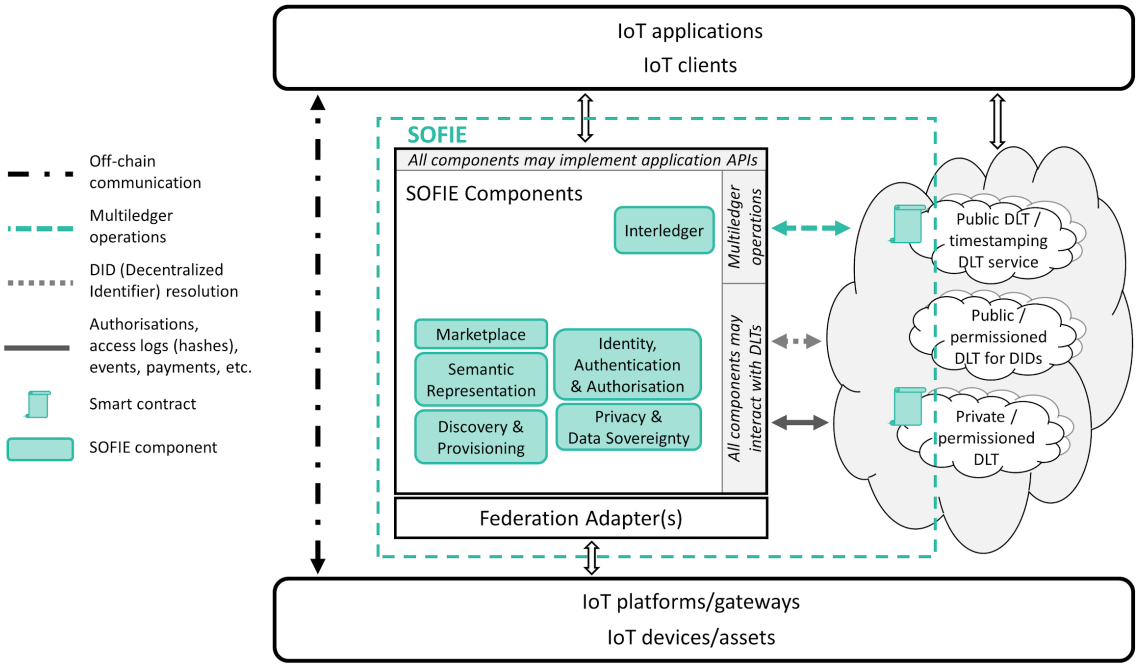

Figure 2.1: The SOFIE framework architecture.

A key element of the SOFIE Architecture, depicted in Figure 2.1, is that it is a *framework/meta architecture* that defines the types of functionalities provided by the components and adapters, but no APIs or implementation for the functionality. This is due to the fact that SOFIE is intended to support IoT federation in many application areas and it is infeasible to define a set of functions that would encompass all the needs (including future needs) of the different application areas. Instead, the Architecture defines key functionalities for federation and

provides example implementations of each component and adapter in the SOFIE Framework. The provided examples are based on the pilots in the SOFIE project and they can be freely adapted and expanded to suit the needs of other applications.

The lowest level of the architecture contains the *IoT systems*: *IoT assets/devices* (or resources) that include, e.g. IoT sensors for sensing the physical environment and actuators for acting on the physical environment, and *IoT platforms/gateways* that include provide additional functionality, e.g. data stores, where the measurements from sensors are collected and made available to third parties.

The *federation adapter(s)* are used to interface the IoT systems with the Architecture. This allows the IoT systems to interact with SOFIE without requiring any changes to the IoT systems themselves. Different scenarios and pilots can utilise different types of federation adapters, which expose only the required parts of the SOFIE functionality to the IoT system and which can implement different protocols, standards etc. depending on the application domain and devices used. The federation adapters used in the SOFIE pilots are discussed in Section 9 and they have been released as open-source software as part of the Framework [Framework].

The main functionality of the SOFIE Architecture is provided by the 6 *components* described in Sections 3-8*:*

- *Interledger* enables secure federation by providing support for atomic transactions spanning two or more ledgers.
- *Identity, Authentication and Authorisation* provides IAA functionalities for the different entities in the system by supporting multiple authentication and authorisation techniques.
- *Privacy and Data Sovereignty* provides mechanisms that enable data sharing in a controlled way and supports privacy preserving surveys using differential privacy techniques.
- *Semantic Representation* is used to enable semantic level interoperability between different IoT systems, services, and data by describing what functions they provide and what interfaces and formats they utilise.
- *Marketplace* allows participants to trade resources by creating auctions, placing offers, and tracking trade completion in a secure, auditable, and decentralised way.
- *Provisioning & Discovery* provides management and discovery of IoT devices, services, and data.

Finally, all the components can expose *application APIs*, which provide the interfaces for IoT clients and applications to interact with the SOFIE components.

Of the six components, the architecture emphasises the *interledger component* (described in Section 3) responsible for interconnecting the different types of decentralised ledger technologies (DLTs), which can have quite different features and functionality. Utilising multiple ledgers that are interconnected through interledger functionality instead of a single DLT for everything provides the flexibility to exploit these trade-offs.

In Figure 2.1 the multiledger operations are positioned next to the Interledger component as it is mostly using that functionality, but any of the other components can also utilise multiledger operations when required. Also, the framework adapters and IoT applications can directly interact with the DLTs, but for simplification this is not shown in the figure.

The open-source implementation was developed over multiple sprints to integrate feedback from pilots in live production environment and utilises Python 3, Node.js, Docker containers and other modern technologies. With this process, the components have reached the

Technology Readiness Level (TRL) of the production environment pilots they have been used in, which means TRL 6 for the Provisioning & Discovery component (used in the CAMG pilot) and TRL7 for all the other components. More details about the TRL levels of each pilot can be found in D5.4 [D5.4].

3 Interledger Component

The main purpose of the SOFIE *Interledger (IL) component* is to enable transactions between actors belonging to different (isolated) IoT platforms or silos. SOFIE assumes that each IoT silo either utilises or is connected to one or more DLTs, with the Interledger component enabling transactions between these DLTs. Several techniques for implementing this type of functionality have been proposed in the literature [Sir2019] and of those the IL component implements a *bridge*-type connection, where activity on one ledger (the Initiator ledger) triggers activity on one or more ledgers (the Responder ledgers) in an atomic manner. The Framework's IL component provides support for connecting different types of DLTs: Ethereum, Hyperledger Fabric, Hyperledger Indy, and KSI; support for additional DLT types can be easily added.

With the Interledger component, it is possible to, e.g. integrate multiple ledgers to a cohesive storage platform that enables the most suitable type of DLT to be used for the type of information at hand and to enable cross-ledger transactions, thus harnessing the individual strengths of the different DLTs. The IL component is used in all SOFIE pilots and the Framework also provides multiple code examples of using the IL including:

- Transferring Data from one ledger to another.
- Storing Data Hashes stores detailed information in a (private) ledger and a hash of the information is then stored in a (public) ledger at suitable intervals using Interledger to benefit from the higher trust of the public ledger.
- Game Asset Transfer implements a state transfer protocol, which is used for managing in-game assets: the assets can either be used in a game or traded between gamers but not both. For both activities a separate ledger is used, and Interledger ensures that each asset is active in only one of the ledgers.
- Hash Time Locked Contracts (HTLCs) describes how to use the Interledger to automate the asset exchange between two ledgers using HTLCs.

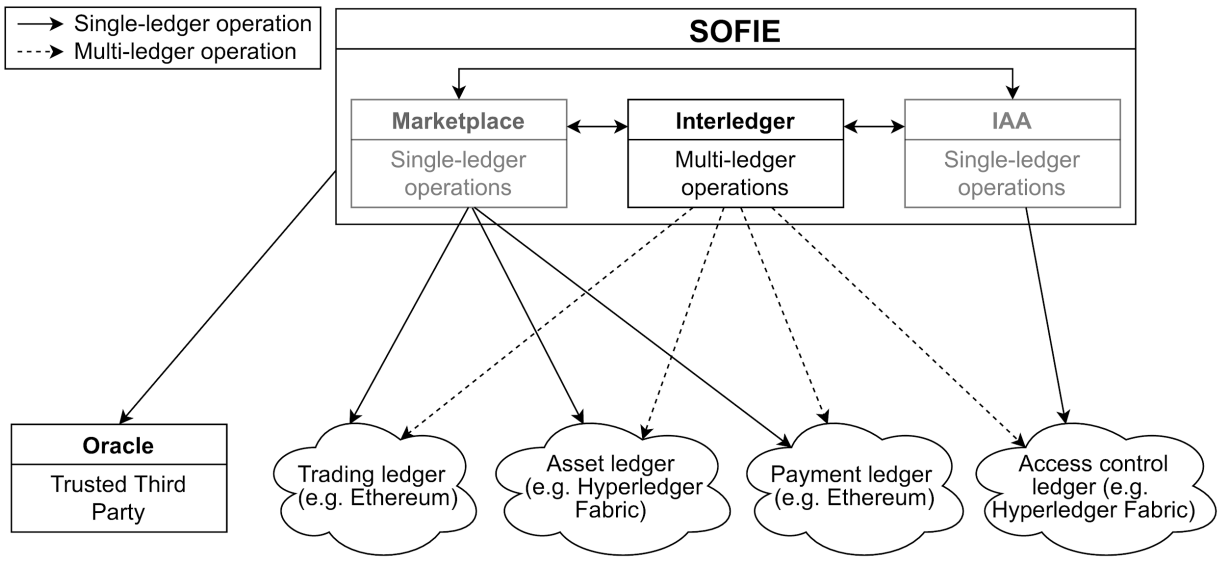

Figure 3.1: Connection between the SOFIE components and the ledgers.

All SOFIE components can directly access the ledgers, but for multi-ledger operations they usually rely on IL as shown in Figure 3.1, where the dashed lines represent multi-ledger operations and continuous lines represent single-ledger operations. In this example, the Asset and Access control ledgers can be permissioned ledgers such as Hyperledger Fabric to reduce costs, while payments and trades can take place on public ledgers such as Ethereum to improve trust. Finally, the *oracle* is a trusted third party in charge of making authoritative statements about the status of some system, e.g. the oracle may track the charging events in the Decentralised Energy Flexibility Marketplace pilot and notify the auction once the agreed amount of energy has been consumed in the correct district. The presence of the oracle and its connections with the ledgers depends on the use case.

3.1 Requirements and Validation

Table 3.1 details the requirements for the IL component (from SOFIE deliverable D2.6).

Table 3.1. Requirements for the Interledger component

Req. ID	Requirement Description	Priority	Category	Source
RF01	User interaction is not required for interledger operations.	MUST	USABILITY	FSC, DEFM, CAMG, S
RF02	There should be support for atomic interledger operations.	SHOULD	SECURITY	CAMG

Table 3.2 details how the component has been validated with the specific tests. All tests are available in the component repository at the specified location and all tests have successfully passed, and with them the component meets all the requirements.

Table 3.2. Validation of SOFIE Interledger component

ID	Validation Process	
RF01	Requirement Description	User interaction is not required for interledger operations.
	Test approach	Functional test
	Test Description	Event on one ledger automatically triggers the transfer of data/asset to another ledger
	Test location	Interledger: tests/system/test_interledger_ethereum.py
RF02	Requirement Description	There should be support for atomic interledger operations.
	Test approach	Functional test
	Test Description	Status of asset transfers is atomic, so that the asset can be accessible only in one ledger
	Test location	Interledger: tests/system/test_interledger_ethereum, solidity/test/tokenTest (testing contract for GameToken)

The IL component has reached TRL 7: it is an integral part of all four SOFIE pilots, including the TRL 7 pilots Food Supply Chain, Decentralised Energy Flexibility Marketplace, and Decentralised Energy Data Exchange as detailed in SOFIE Deliverable D5.4 [D5.4].

To ensure sufficient scalability, the IL component has been designed to handle large volumes of traffic: since there is no processing of data payloads off-ledger, the overhead of the component is marginal compared to the latency of the transactions on the distributed ledgers themselves. As an example, when moving assets from one ledger to another in the Context-Aware Mobile Gaming pilot, the off-ledger processing of the IL component takes only 0.05s out of the total latency of 3.07s (when using private Ethereum ledgers), and the on-ledger latency on a public ledger could be significantly higher due to the longer block time, thus making the off-ledger time negligible for those cases. Also, it is possible to increase the IL throughput with almost no limit, simply by running multiple IL nodes in parallel so that each node manages a different set of connections.

3.2 Services and Interfaces

The IL component is used by configuring *Interledger instances* between the ledgers: each instance is a unidirectional connection between one Initiator and one or more Responders. It is also possible to enable bidirectional communication between two ledgers by configuring two Interledger instances in opposite directions, but multiple Initiators for a transaction are not supported due to the application specific semantics of such operations. However, applications can implement such functionality e.g. using smart contracts.

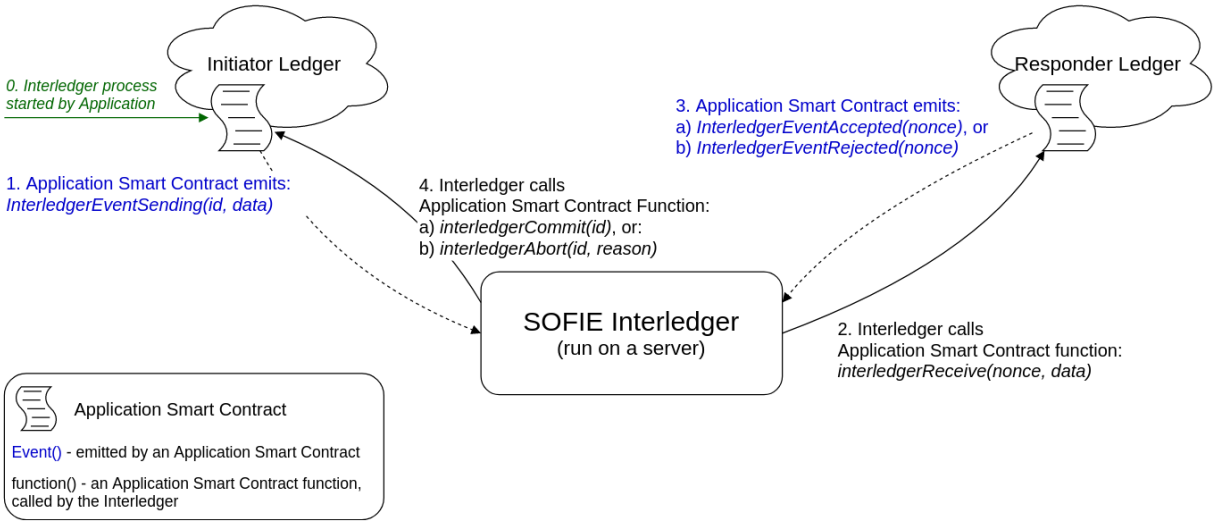

Figure 3.2: An Interledger Instance has been configured between application smart contracts on Initiator and Responder ledgers.

As shown in Figure 3.2, an application wishing to utilise IL has first deployed two Smart Contracts (SC) that implement the Sender and Receiver interfaces on the Initiator and Responder ledgers (detailed in Table 3.3), respectively, and then configured the Interledger instance between them. The IL component now listens for *InterledgerEventSending*-events from the configured Initiator ledger SC, and when one is emitted (1), it passes the *data* parameter to the configured SC on the Responder ledger(s) by calling the *interledgerReceive* function (2). The application SC on the Responder ledger then either has to accept or reject the call by emitting the corresponding event (*InterledgerEventAccept* or

InterledgerEventReject) (3), and that reply is then conveyed to the Initiator ledger's SC by calling the corresponding function (*InterledgerCommit* or *InterledgerAbort*) (4) thus completing the transaction. In step 2, the application smart contract also includes an application chosen *id* parameter, which is only used between the SC and IL (it is not passed to the Responder ledger). The *id* does not need to be globally unique, so the application can e.g. use it to group certain types of events or use unique values to distinguish each transaction. On the Responder side, an IL-generated *nonce* is used to track each transaction, and it is mapped back to the original *id* for step 4. This example uses Ethereum ledgers as Initiator and Responder. Depending on the ledger used, the details of invoking IL etc. may differ as not all ledgers support smart contracts, events etc. Hyperledger Indy and KSI are examples of such ledgers, so their implementations in the Framework can be used as examples when adding support for similar ledgers.

Table 3.3. Interfaces of the IL component

ID	Interface Content	
IF01	Name	**Sender Interface**
	Description	Used by the application smart contract on the Initiator ledger to trigger IL transactions
	Key inputs	Transaction data
	Response	Transaction success/failure status
IF02	Name	**Receiver Interface**
	Description	Used by the application smart contract on the Responder ledger to receive the transaction
	Key inputs	Transaction data
	Response	Transaction success/failure status

3.3 The internal structure

The high level structure of the IL component is shown in Figure 3.3. The logic of passing the transactions back and forth and maintaining the related state information is implemented in the *IL Core,* while the support for the different types of DLTs is enabled through the use of stateless *ledger-specific adapters*, which enable the DLT to act as Initiator and/or Responder. The Framework's IL component provides adapters with both roles for Ethereum and Hyperledger Fabric, Initiator support for Hyperledger Indy, and Responder support for KSI.

The IL Core does not perform any processing on the *data* parameter it passes through, but the adapters can implement such functionality if it is relevant for the DLT in question as has been done with the KSI Responder adapter, which automatically calculates a hash of the *data* parameter and stores only the hash to KSI. Typically, though, adapters only pass through the information.

The Interledger Instance will operate in one of the two alternative modes depending on whether the Instance is between one Initiator and one Responder (one-on-one mode) or one Initiator and multiple Responder (multi-ledger mode). The multi-ledger mode has two sub

modes: either all (all-N) or at least *k* of the N (k-out-of-N) Responders have to accept the transaction or the whole transaction is rolled back.

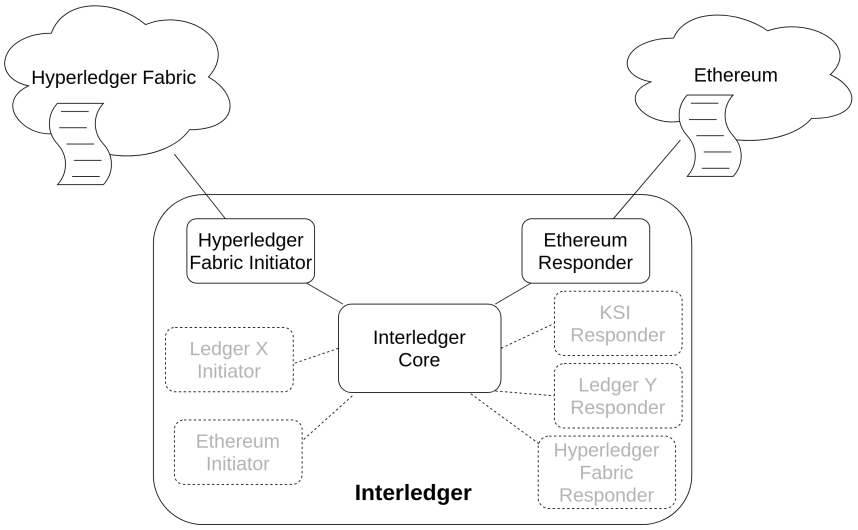

Figure 3.3: Functional overview of the SOFIE IL component.

The interactions in the *one-on-one mode* is shown in Figure 3.4. The Initiator adapter is initialised with the *listen_for_events* function (1) after which it starts catching the *InterledgerEventSending* events from an Initiator ledger (2). For every caught event the adapter calls the Core's *send_tranfer* function (3), which generates a *nonce* to replace the *id* parameter, stores all related data in an internal state structure, and finally calls the *send_data* function of the Responder adapter (4). The adapter then calls the application SC function *interledgerReceive* (5). The application then either accepts or rejects the transaction by emitting either the *InterledgerEventAccepter* or *InterledgerEventRejected* event (6). The Responder adapter passes the reply to the Core using the *process_result* function (7), which maps the *nonce* back to the original *id* and calls the Initiator adapter's *commit_sending* or *abort_sending* function depending on the reply (8). The Initiator adapter completes the transaction by calling either the *InterledgerCommit* or *InterledgerAbort* of the application smart contract (9).

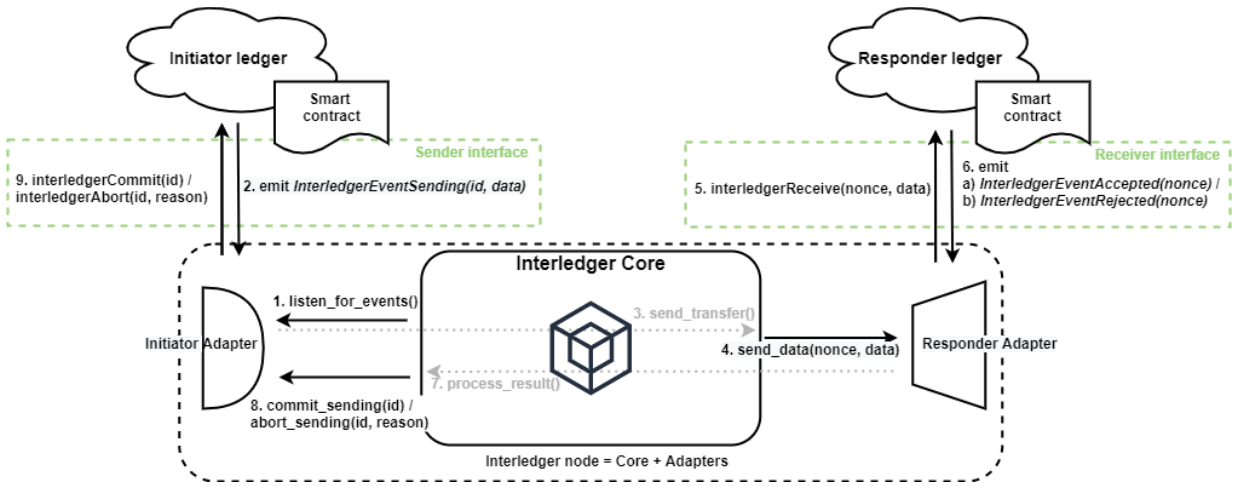

Figure 3.4: Interledger in one-on-one mode.

In the *multi-ledger mode shown in Figure 3.5* all steps on the Initiator side (1-2 and 12-13) remain the same, but the Responder side has some additional steps: it now implements a *two-phase commit* to ensure sufficiently many application SCs on the Responder ledgers accept the transaction. So the Core in step 4 calls the *send_data_inquire* function of each N Responders, which passes the inquiry to the application SC with the *interledgerInquire* function (5). After the application SC's reply with either *InterledgerInquiryAccepted* or *InterledgerInquiryRejected* event (6), the Responder adapter then calls the Core's *transfer_inquiry* function (7). After all the Responders have replied, the Core will either commit (adapter's *send_data function*) or roll back the transaction (*abort_send_data* function) (8). The Responder adapter will then pass the information to the application SCs with either *InterledgerReceive* or *InterledgerReceiveAbort* function (9). The application SCs will confirm by emitting either *InterledgerEventAccepted* or *InterledgerEventRejected* event (10), and the adapter will pass the information to the Core with *process_result* function (11), after which the transaction concludes as in the one-on-one mode transaction.

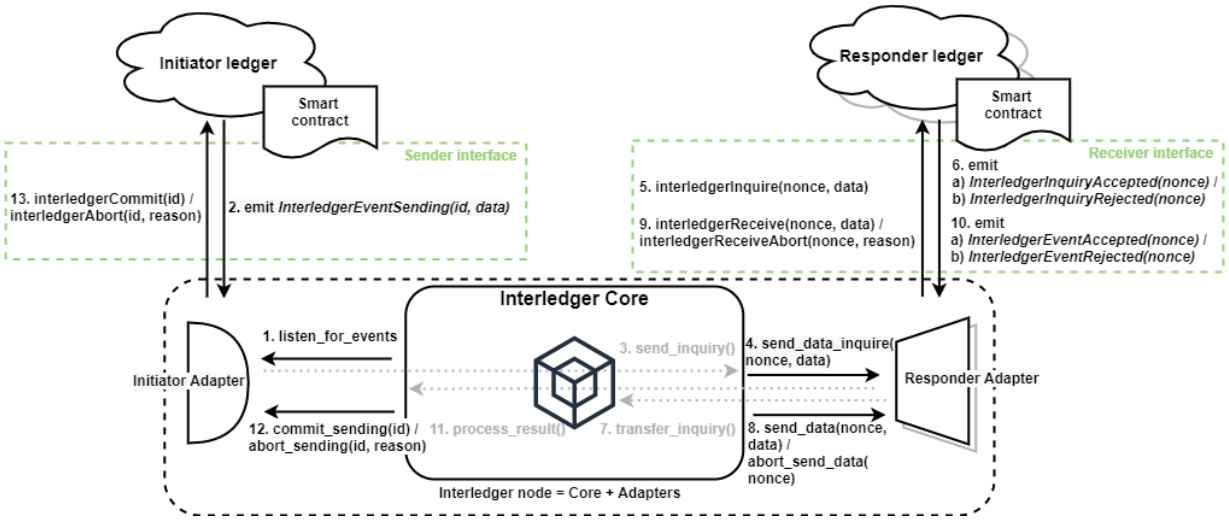

Figure 3.5 Interledger in Multi-ledger mode.

Since the IL component communicates with multiple DLTs, it abstracts each ledger API (e.g. Web3 for Ethereum[1]) in order to be reusable as shown in Figure 3.6.

The basic IL component is run on a single computer (single-node IL), which requires the applications and their users to trust the party running the node. The Framework also provides an initial version of a Decentralised Interledger (DIL), where a consortium of parties are jointly responsible for the IL services. Each member runs one or more IL nodes all of which are coordinated using a *shared state*. This reduces the trust required as now the users of the IL services only need to trust the consortium as a whole, as well as improves throughput and resiliency of IL. Switching to DIL in no way affects the external interfaces or use of the IL service, only the internal operations of IL.

[1] https://web3py.readthedocs.io/en/stable/

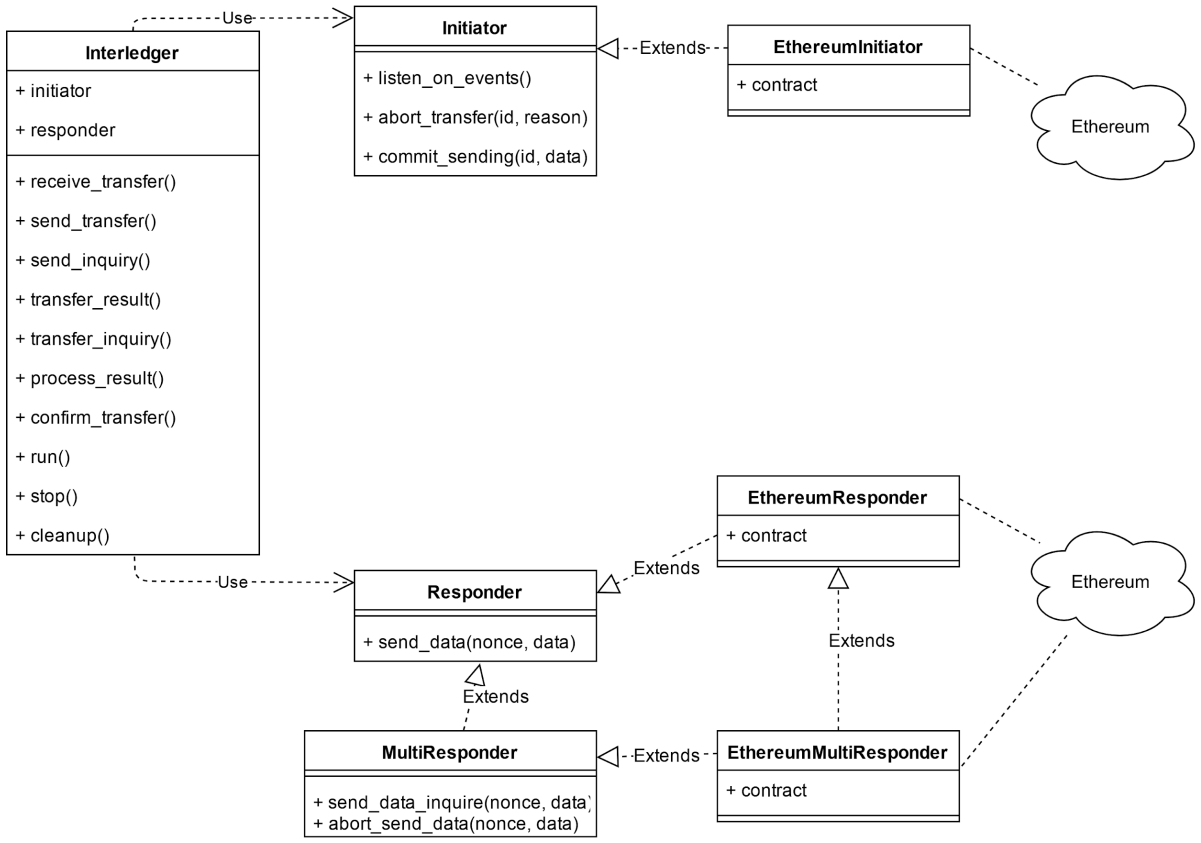

Figure 3.6: The Interledger component's internal structure.

Overall, the DIL architecture remains very similar to the single node IL as all the changes required to support the DIL functionality only affect the IL Core, and all the ledger adapters and application smart contract remain the same. The major change from single node IL is that instead of the Core handing all transactions to the correct ledger adapter on the same node it now writes the information to the shared state that keeps a shared records of transfers for all DIL nodes in the consortium and the node responsible for continuing the transaction (together with standbys) is then chosen in a pseudo-random manner (by using a deterministic algorithm and a hash function as the source of randomness). Further redundancy can be provided by having all capable nodes (nodes may have different capabilities and not all nodes necessarily can operate with all ledgers) monitor the related ledgers and competing to update the shared state.

The architecture and transaction flow of DIL are detailed in Figure 3.7. First, all capable nodes listen for events on the Initiator ledger (1) and compete to write it to the shared state (2) and then check that the successful write is correct. Using a deterministic algorithm a pseudo-randomly selected node is chosen to perform the Responder operations while others are chosen as backups (3). As the Responder operation may be a multi-ledger operation that can take a while to complete, the chosen Responder will first signal that it will proceed with the operation (4) and then proceed with the Responder operations of the transaction (5-6, multi-ledger operations will have further steps due to the two-phase commit), and once complete, it will write the result to the shared ledger (7) (if steps 4 or 7 time out, the first backup node will take over steps 4-7 etc.). A validator node is chosen using the same algorithm (8), but this time it will not write a separate acceptance to the shared ledger as the

Initiator operation is much faster to complete involving only submitting the result to the Initiator ledger (9) and updating the shared status (10-11) (again, if step 11 times out, the first backup will step in etc.).

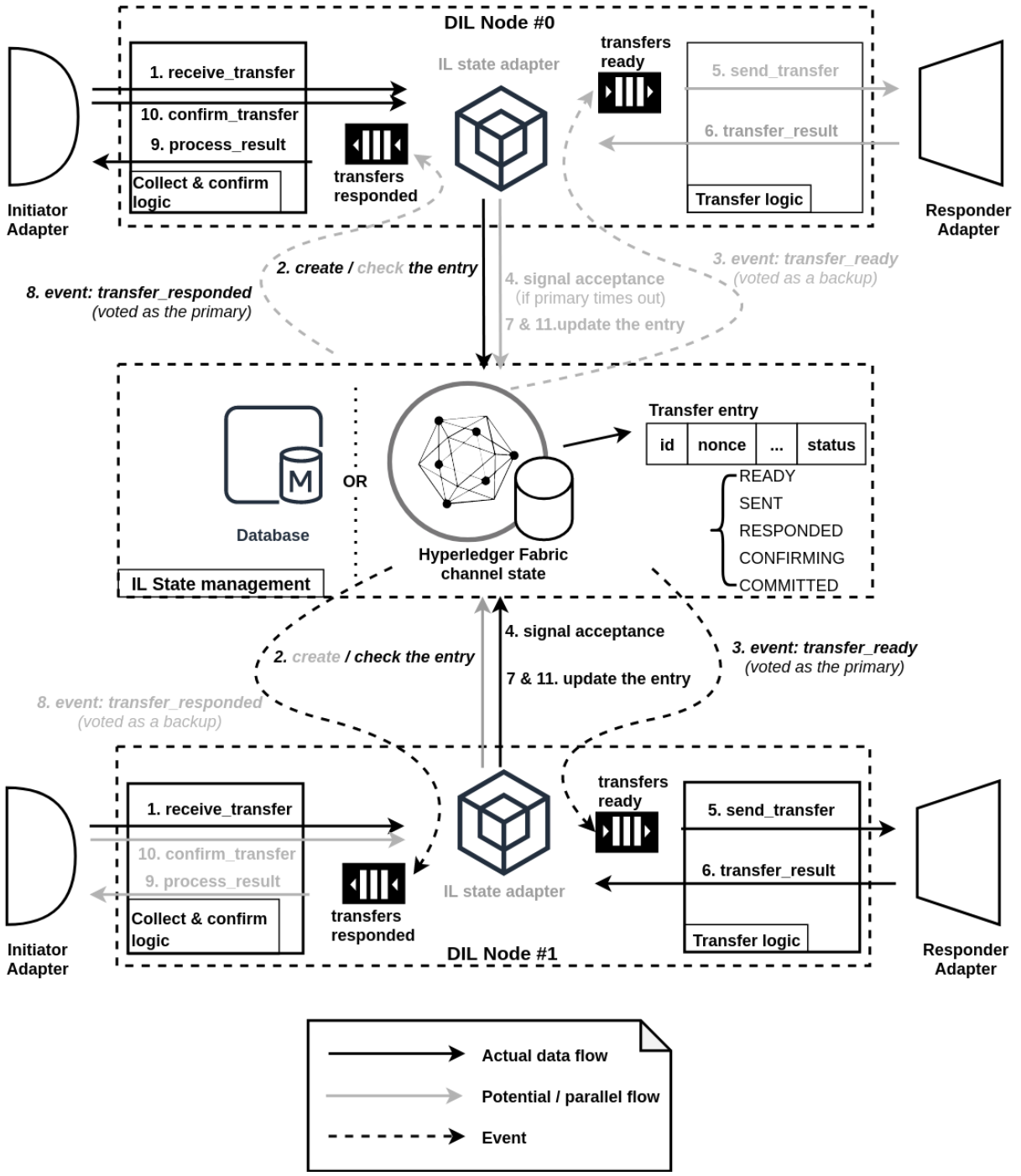

Figure 3.7: Architecture of the Decentralised Interledger (DIL), where a shared state is used to coordinate multiple IL nodes.

4 Identity, Authentication and Authorisation Component

The goal of the *Identity, Authentication and Authorisation* (IAA) component is to provide mechanisms that can be used for authenticating and authorising users wishing to access a protected resource.

User authentication and authorisation is implemented using *access tokens*. Tokens can be regarded as JSON-encoded data structures that include *properties* about users. These properties are then validated by the IAA component in order to decide which resources a user can access. In its present form, the IAA component can use the following types of access tokens: Hyperledger Indy Decentralised Identifiers (DIDs) and Verifiable Credentials (VC), W3C-based VC, JSON web tokens (JWT), and JWT backed by Ethereum ERC-721 tokens (as described in [Fot2020]).

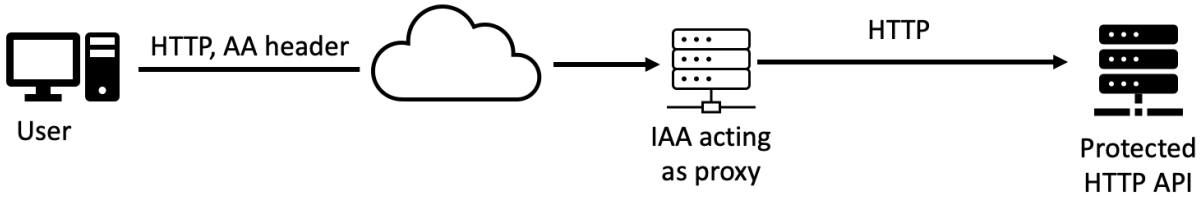

Figure 4.1: IAA Overview

The IAA component, as it can be seen in Figure 4.1, acts as an HTTP forward proxy and intervenes in the communication between users and protected resources. Users include their access token in their HTTP resource access request as an HTTP header, which IAA parses and validates; If the token is valid, the IAA forwards the request to the protected resource. One of the main advantages of IAA is that it is transparent both to end-users and protected resources.

Token validation is based on access control policies included in the IAA configuration file. From a high level perspective these policies define the type of tokens and the properties a token must include, so that a user can receive permission to access a resource. Therefore, using these policies, administrators can implement role-based access control, Access control policies are implemented as *filters* that are used for parsing the token. Filters, examples of which are included in the *conf/iaa.conf* file in the IAA GitHub repository, may define rules for accepted token issuers including required properties, token structure, and others..

4.1 Requirements and Validation

Table 4.1 summarises the requirements for the IAA component (from D2.6).

Table 4.1. Requirements for the IAA component.

Req. ID	Requirement Description	Priority	Category	Source
RF03	Resource owners must be able to delegate the authentication and authorisation tasks for their resources.	MUST	OPERATIONAL	FSC, S
RF04	The IAA component must provide users the capability to revoke authorisations.	MUST	SECURITY	FSC

RF05	The IAA component must allow individuals to control their personal information and digital identities (e.g. support self-sovereign identity technology).	MUST	SECURITY	DoA
RF06	The IAA component must support secure, tamper-proof, and verifiable logging of transactions and events.	MUST	SECURITY	FSC
RF07	The IAA component must support Role Based Access Control (RBAC).	MUST	SECURITY	FSC
RF08	Cryptographic algorithms used by SOFIE should be open-source, transparent, and as independent as possible of any particular architecture.	SHOULD	AUDITABILITY	FSC, S
RF09	SOFIE should support the execution of authorisation and authentication functionality on devices with constrained processing, storage, battery, and network connectivity.	SHOULD	OPERATIONAL	S

Table 4.2 details how the component has been validated with the specific tests. All tests are available in the component repository at the specified location and all tests have successfully passed, and with them the component meets all the requirements.

Table 4.2. IAA validation results

RF03	*Requirement Description*	Resource owners must be able to delegate the authentication and authorisation tasks for their resources.
	Test approach	Documentation
	Test Description	The IAA component can be configured to accept a wide range of tokens issued by 3rd party authorization servers.
	Test location	IAA's repository documentation, "Configuration" chapter
RF04	*Requirement Description*	The IAA component must provide users the capability to revoke authorisations.
	Test approach	Functional test
	Test Description	This test involves an authorization server implemented by the PDS component (see next section). The authorization server creates a token and logs it in an ERC-721 smart contract. Then the authorization server marks this token as *revoked*. A client application performs a resource access request which is proxied through IAA. IAA retrieves the status of the token from the ERC-721 smart contract, it detects that the token has been revoked, and rejects it.
	Test location	IAA tests/test_erc721.py

RF05	*Requirement Description*	The IAA component must allow individuals to control their personal information and digital identities (e.g. support self-sovereign identity technology).
	Test approach	Functional test
	Test Description	The test uses a script that emulates a user. The script is configured with a valid DID and a valid VC. The VC includes a number of user properties. The script sends a resource access request to IAA which responds with a challenge that requests the user script to "reveal" some of the properties included in the VC . The user script responds to the challenge with an appropriate *Verifiable Presentation.*
	Test location	IAA tests/test_indy_agent.py
RF06	*Requirement Description*	The IAA component must support secure, tamper-proof, and verifiable logging of transactions and events.
	Test approach	Functional test
	Test Description	The test uses a script that emulates a user. The user script is configured with a valid JWT token. The user script performs a resource access request and includes in the request the token. The request is received by IAA which logs the token in an Ethereum smart contract.
	Test location	IAA test/test_logging.py
RF07	*Requirement Description*	The IAA component must support Role Based Access Control (RBAC).
	Test approach	Documentation
	Test Description	RBAC is implemented with the use of VCs. IAA can be used to verify a VC.
	Test location	IAA's repository documentation, "Examples" chapter
RF08	*Requirement Description*	Cryptographic algorithms used by SOFIE should be open-source, transparent, and as independent as possible of any particular architecture.
	Test approach	Documentation
	Test Description	IAA supports standardised cryptographic algorithms.
	Test location	IAA's repository documentation, "Key technologies" chapter
RF09	*Requirement Description*	SOFIE should support the execution of authorisation and authentication functionality on devices with constrained processing, storage, battery, and network connectivity.
	Test approach	Functional test
	Test Description	The test pre-configures IAA with the DID document of a Hyperledger Indy DID. Therefore, IAA does not have to retrieve it from an Indy blockchain. A user script performs a resource access request, using that DID as an access

		token. Then IAA authenticates this DID using only local information, and without needing network connectivity.
	Test location	IAA test/test_indy_api.py

The IAA component has reached TRL 7: it is an integral part of two SOFIE pilots on TRL 7, Food Supply Chain and Decentralised Energy Data Exchange as detailed in SOFIE Deliverable D5.4 [D5.4].

4.2 Services and Interfaces

The IAA component provides the following 4 interfaces as shown in Figure 4.2 and detailed in Table 4.3:

- **IAA Proxy:** This is an HTTP Proxy exposed by the component. It can be accessed directly by a client and its goal is to provide client authentication and authorisation.
- **Blockchain I/O:** This interface is responsible for interacting with ledgers (or the interledger component), in order to read blockchain-based tokens.
- **Internal I/O:** This interface is used by the component for interacting with the protected (HTTP) services
- **Administrative interface:** This interface is used for configuring the component with the appropriate policies.

The goals of the interfaces are: (i) to enable generic client applications or other components to access the IAA operations without significant effort, (ii) to allow protection of arbitrary HTTP-based resources, and (iii) to facilitate rich access control policies.

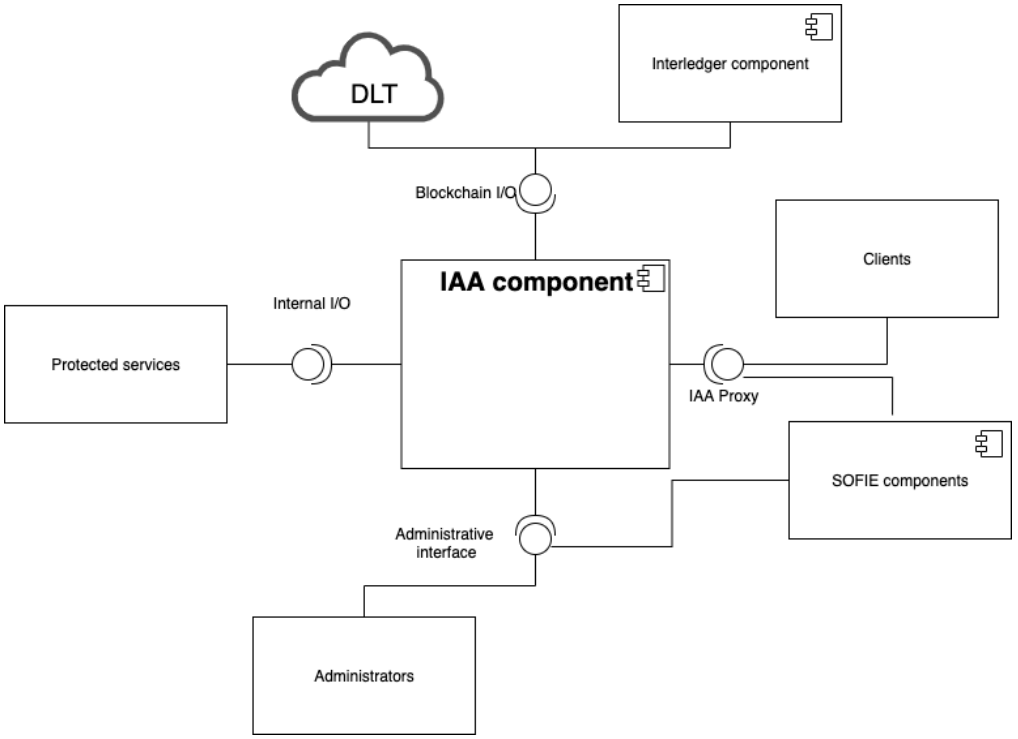

Figure 4.2: The IAA component's interfaces.

Table 4.3. Interfaces of the component.

ID	Interface Content	
IF01	Name	**IAA Proxy**
	Description	It is used by clients to access a protected resource
	Key inputs	Access token
	Response	The protected resource response, or an HTTP 401 error
IF02	Name	**Blockchain I/O**
	Description	It is used by the component for reading ERC-721 based tokens stored in an Ethereum smart contract
	Key inputs	Token Id
	Response	Hash of the token, Ethreum address of the token owner
IF03	Name	**Internal I/O**
	Description	It is used by the IAA component for communicating with the protected resources
	Key inputs	The HTTP request of an authorised user
	Response	The HTTP response of the protected resource
IF04	Name	**Administrative interface**
	Description	It is used for configuring the IAA component with the proper access control policies
	Key inputs	A JSON-encoded configuration file
	Response	No output

4.3 The internal structure

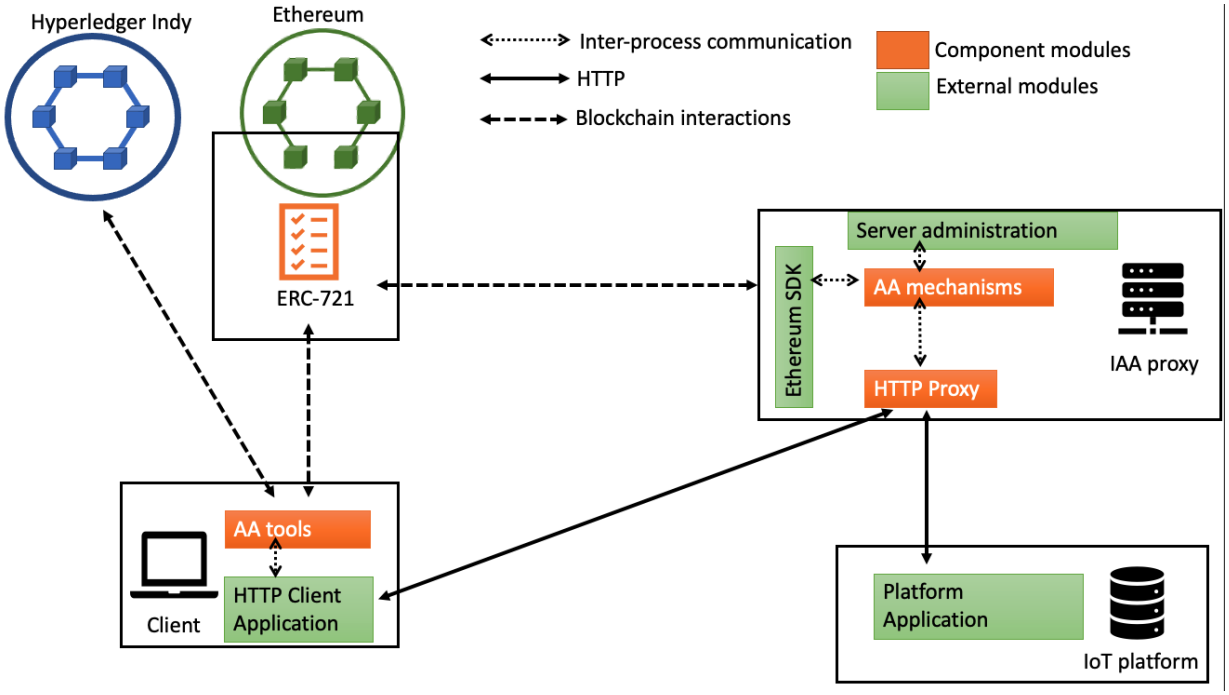

Figure 4.3: IAA component internal structure.

As depicted in Figure 4.3, the IAA component is composed of 3 entities: Client modules, Blockchain modules, and the IAA proxy.

Client modules
This entity includes modules that can be used for generating HTTP authentication headers.

Blockchain modules
The IAA component includes tools for interacting with Hyperledger Indy as well as a smart contract that implements the functionality required by OAuth2-based authorisation using blockchains, as described in [Fot2020].

IAA proxy
This entity is responsible for proxying HTTP requests to protected IoT platforms. It includes a number of Authentication and Authorisation mechanisms.

Operations
The main operations related to the IAA component are illustrated in Figure 4.4.

Initially, a client generates an HTTP request, which includes the access token in the appropriate HTTP header. Furthermore, and depending on the type of the token, it may include a *proof-of-possession*, using DPoP[2]. Then it sends the request.

The HTTP request is received by the IAA proxy. The proxy extracts the access token and validates it. The validation process is based on the proxy configuration file. Depending on the token type, the proxy may use additional information provided by the IAA smart contract, i.e., an Ethereum smart contract that implements the functionality described in [Fot2020].

If the validation process is successful, the IAA proxy removes the authentication header and forwards the request to the protected resource; the resource responds and the proxy forwards the response back to the client.

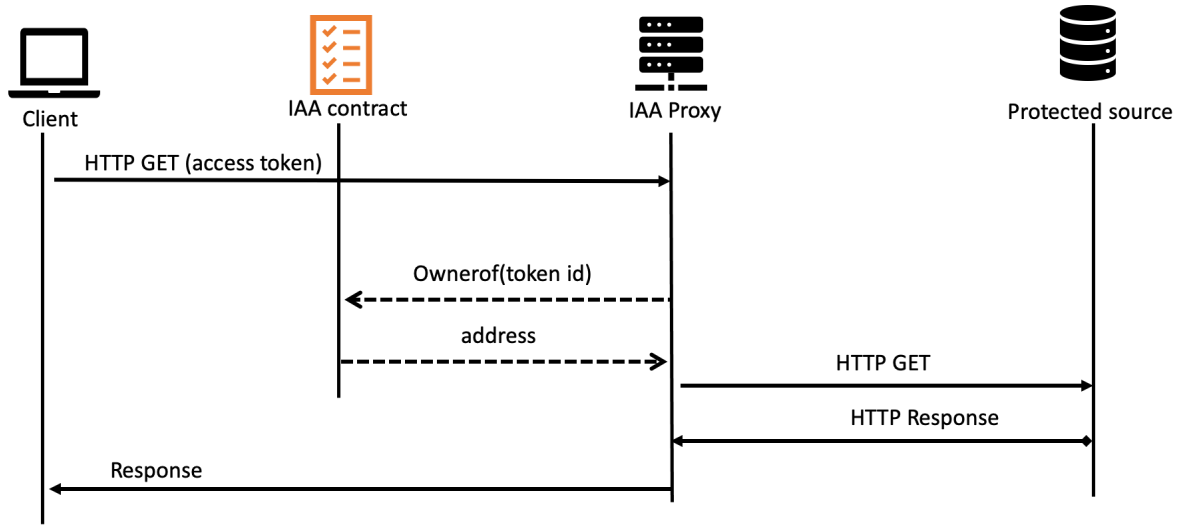

Figure 4.4: IAA component flowchart.

[2] https://tools.ietf.org/html/draft-ietf-oauth-dpop-02

5 Privacy and Data Sovereignty Component

The Privacy and Data Sovereignty (PDS) Component is composed of two modules: the Privacy module and the Data Sovereignty module. Each module can be used independently from the other.

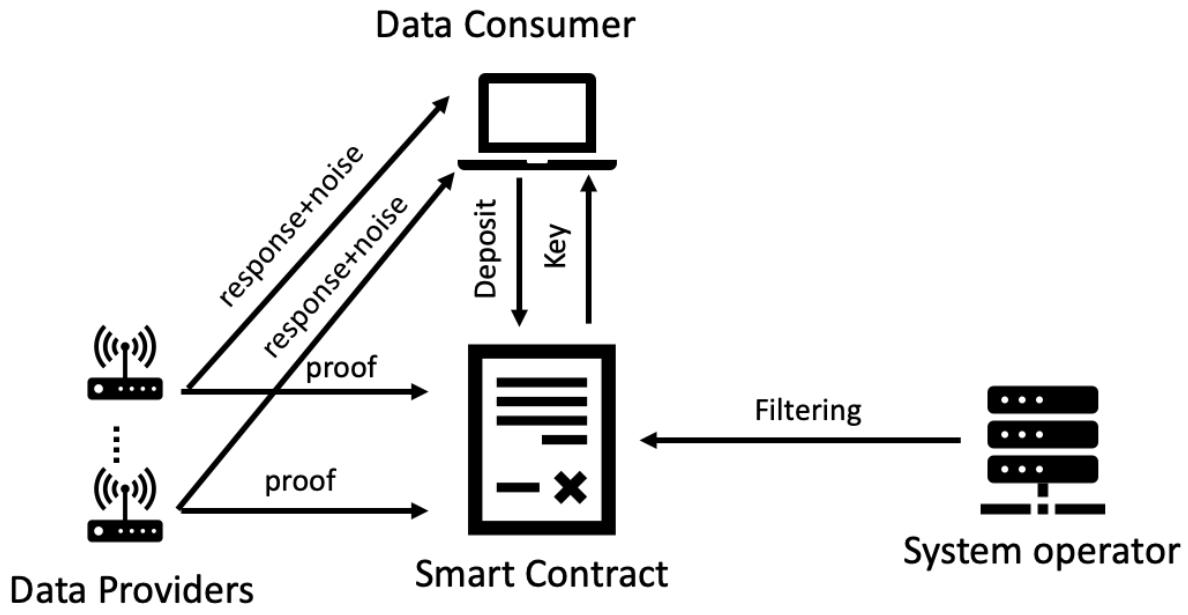

Figure 5.1: Privacy module overview.

The Privacy module enables the creation of *privacy preserving surveys*. These are surveys that allow users to add *noise* to their responses using local differential privacy mechanisms. The addition of the noise prevents 3rd parties from learning meaningful information about specific users, but at the same time aggregated statistics can be extracted. The accuracy of the extracted statistics increases with the number of responses. An overview of the privacy module is illustrated in Figure 5.1. From a high level perspective, the privacy module allows *data consumers* to *buy* noisy responses from *data providers*. This process creates the following requirements: (i) responses must be filtered (by a *system operator* so as to mach some criteria set by the data consumer, (ii) a data consumer should only pay for the service if an appropriate number of responses has been collected, and (iii) a data consumer should not be able to extract any statistics before it pays. All these requirements are accommodated using a smart contract.

The Data Sovereignty module implements an OAuth 2.0 Authorisation Server. This server accepts *authorisation grants* and if the grant is valid it generates an *access token* encoded using the JSON web token (JWT) format. Accepted types of authorisation grants are: Decentralised Identifiers (DIDs), Verifiable Credentials (VCs), and pre-shared secret keys. The generated access token can be used by any web service, as well as with SOFIE's Identity, Authentication, and Authorisation (IAA) Component.

5.1 Requirements and Validation

Table 5.1 below summarises the requirements for the PDS component (from deliverable D2.6).

Table 5.1. Requirements for the PDS component.

Req. ID	Requirement Description	Priority	Category	Source
RF10	SOFIE must follow the data minimisation principle for personal data and only request or process what is necessary for the situation and purpose.	MUST	OPERATIONAL	FSC
RF11	Processing of individual's personal data is justified by a valid legal basis, e.g. a valid consent from the individual.	MUST	POLICY & REGULATION	L
RF12	Consent to process personal data must be revocable at any time.	MUST	POLICY & REGULATION	L
RF13	SOFIE must allow organisations and actors to manage (create, update, delete) their own data privacy policies.	MUST	POLICY & REGULATION	FSC, S
RF14	SOFIE should support user privacy even when aggregate statistics are made public (e.g. using differential privacy mechanisms).	SHOULD	POLICY & REGULATION	DoA

Table 5.2 details how the component has been validated with the specific tests. All tests are available in the component repository at the specified location and all tests have successfully passed, and with them the component meets all the requirements.

Table 5.2. PDS validation results.

RF10	*Requirement Description*	SOFIE must follow the data minimisation principle for personal data and only request or process what is necessary for the situation and purpose.
	Test approach	Documentation
	Test Description	PDS can be configured with a specific proof request
	Test location	PDS's repository documentation, "Configuration" chapter
RF11	*Requirement Description*	Processing of an individual's personal data is justified by a valid legal basis, e.g. a valid consent from the individual.
	Test approach	Functional test
	Test Description	The test is configured with a valid VC. The test invokes the VC verification, which generates a proof request. The test generates the proof and outputs the verification result.
	Test location	PDS tests/test_indy_agent.py
RF12	*Requirement Description*	Consent to process personal data must be revocable at any time.

	Test approach	Documentation
	Test Description	The documentation described how to set an expiration time on a VC
	Test location	PDS's repository documentation, "Examples" chapter
RF13	*Requirement Description*	SOFIE must allow organisations and actors to manage (create, update, delete) their own data privacy policies.
	Test approach	Documentation
	Test Description	PDS can be configured with arbitrary VC schemas.
	Test location	PDS's repository documentation, "Configuration" chapter
RF14	*Requirement Description*	SOFIE should support user privacy even when aggregate statistics are made public (e.g. using differential privacy mechanisms).
	Test approach	Documentation
	Test Description	PDS can be configured to apply RAPPOR local differential privacy mechanism.
	Test location	tests/privacy/test_privacy.py

The PDS component has reached TRL 7: it is an integral part of two SOFIE pilots on TRL 7, Food Supply Chain and Decentralised Energy Data Exchange as detailed in SOFIE Deliverable D5.4 [D5.4].

5.2 Services and Interfaces

The Data sovereignty module provides the following 3 interfaces:

- **PDS API:** This is the HTTP REST API of the module's OAuth2.0 *Authorisation Server*, which can be accessed directly by a client. It receives as input an *authorisation grant* and it outputs an *access token*.
- **Blockchain I/O:** This interface is responsible for interacting with ledgers (or the interledger component), in order to record tokens as well as audit logs.
- **Administrative interface:** This interface is used for configuring the module with the appropriate policies.

The Privacy module provides the following 2 interfaces:

- **Blockchain I/O:** This interface is responsible for interacting with ledgers (or the interledger component), in order to record surveys and responses proofs.
- **Administrative interface:** This interface is used for configuring the module with the appropriate cryptographic keys.

The interfaces have been depicted in Figure 5.2 and detailed in Table 5.3.

The goals of the interfaces listed above are: (i) to enable generic client applications or other components to access the PDS operations without significant effort, (ii) to allow fair exchange of statistics and tokens, and (iii) to facilitate access token generation rules.

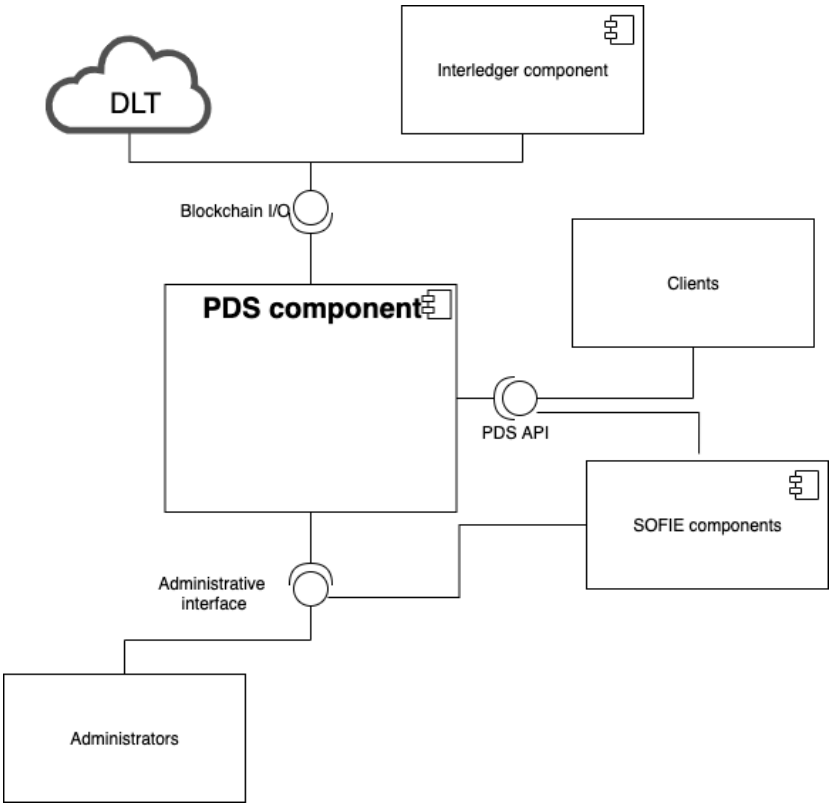

Figure 5.2: The PDS component's interfaces.

Table 5.3. Interfaces of the component.

ID	Interface Content	
IF01	Name	**PDS API**
	Description	Used for accessing the OAuth2.0 authorisation server, implemented by the PDS component
	Key inputs	Authorisation grant
	Response	Access token
IF02	Name	**Blockchain I/O**
	Description	This interface is used for recording blockchain-based tokens, as well as for logging access tokens to a smart contract-based log. Additionally, it is used for recording privacy-preserving surveys, as well as proofs of responses.
	Key inputs	ERC721 token, encrypted token, survey configuration, hash of a survey response
	Response	Record summary
IF03	Name	**Administrative interface**
	Description	This interface is used for configuring the PDS component with rules for validating access tokens, as well as with parameters for the local differential mechanism
	Key inputs	A JSON-encoded configuration file
	Response	No output

5.3 The internal structure

The Privacy and Data Sovereignty component is composed of the following 3 entities:

Client modules
This entity includes modules that can be used for obtaining an access token using OAuth2.0, as well as for adding local differential noise to survey responses.

Blockchain modules
The PDS component includes tools for interacting with Hyperledger Indy, a smart contract that implements the functionality required by OAuth2-based authorisation using blockchains (this process is described in [Fot2020]), a smart contract used logging tokens, as well as a smart contract for recording proofs of responses to surveys.

PDS authorisation server
This entity implements an OAuth2.0 authorisation server.

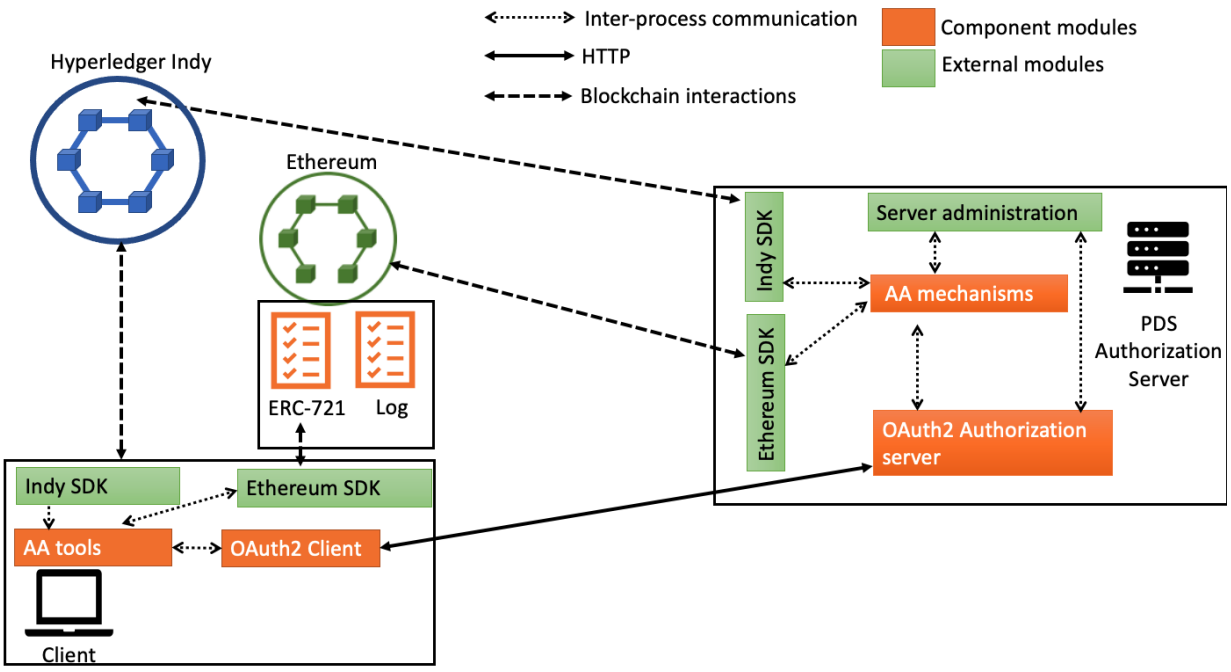

Figure 5.3: Privacy and Data Sovereignty component internal structure.

Operations
The Data sovereignty module implements OAuth 2.0 specifications, and in particular it implements section 4.4 of RFC6749 *Client credentials grant*. The only difference is that is adds three optional fields:

Field	*Semantics*
log-token	Instructs the authorisation server to record the generated access token in an Ethereum smart contract
erc-721	Instructs the authorisation server to create a supplementary ERC-721 based token
enc-key	Instructs the authorisation server to encrypt the generated token using the provided encryption key.

The options *log-token* and *enc-key* should be combined, otherwise the token will be recorded in the blockchain in plaintext.

The privacy module allows data providers to send to data consumers responses to surveys that are protected using local differential privacy. These responses are filtered by the system operator based on criteria specified by the data consumer. The system operator and the data providers share a secret key (PSK). PDS implements the following protocols (see also figure 5.4), using the following notation:

Symbol	*Meaning*
E(k,m)	Symmetric encryption of message *m*, using the key *k*
H(m)	Hash of message *m*
HMAC(k,m)	Keyed MAC of message *m* using key *k*

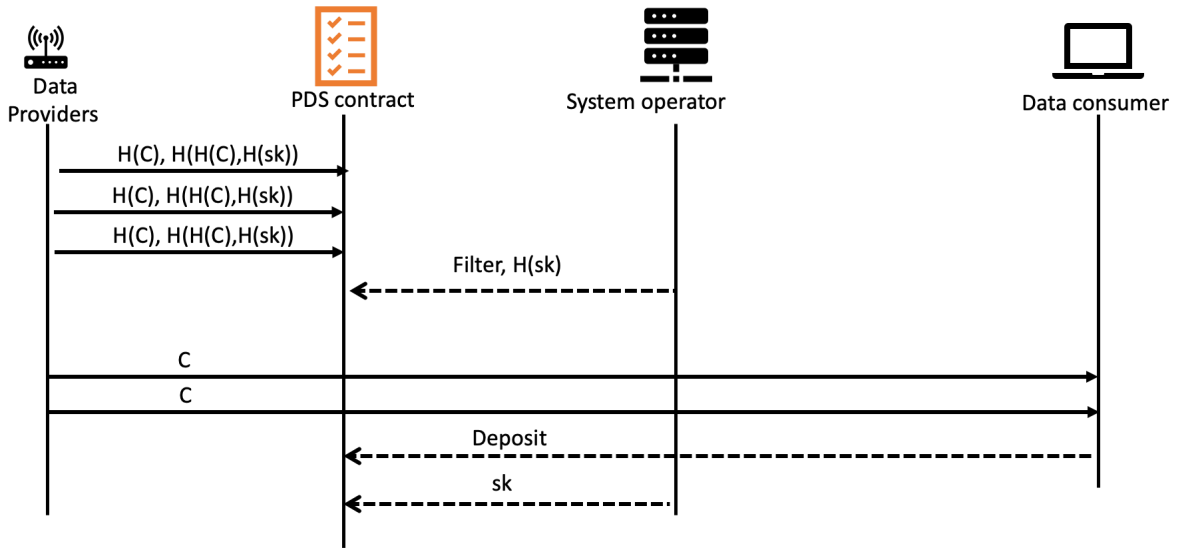

Figure 5.4: Privacy module overview. Initially 3 data providers commit a response. The system operator approves 2 of them.

Survey setup

With this protocol, a system operator and a data consumer agree on a smart contract that includes: (i) the question that data providers should respond to, (ii) filtering rules, (iii) the number of responses that should be collected, (iv) an amount of digital currency each responder will receive, and (v) a service fee. Additionally, the system operator and the data consumer agree on a nonce *n*: all data providers (and the system operator) can derive an encryption key sk = HMAC(psk,n). Then, the system operator sends H(sk) to the data consumer. Finally, n is included in the smart contract.

Response commit

Any provider wishing to participate in the survey, prepares a *noisy* response *R* using local differential privacy. It derives sk using the nonce n included in the smart contact and generates a ciphertext C=E(sk,R). Then it records in the smart contract H(C), as well as H(H(C),H(sk)). Since a data consumer knows both H(C) (from the smart contract) and H(sk) (from the service operator) it can also calculate the second hash and verify that the data provider derived the correct key.

Response filtering

The system operator indicates in the smart contract which of the providers that responded abide by the agreed filtering criteria. If the number of the (valid) responders is bigger than the number of the responses that should be collected, then the system operator indicates which of the responders will be compensated. It also reveals H(sk). All approved data providers send to the consumer the encrypted, noisy responses (i.e., C=E(sk,R) calculated with the Response Commit protocol)

Fair exchange

The data consumer verifies the integrity of the received encrypted responses with the hash stored in the smart contract. When it receives the agreed number of responses it deposits to the smart contract the appropriate amount of currency.Then the system operator reveals sk; if the hash of sk matches the one provided with the response filtering, the contract transfers the deposit to the appropriate entities.

6 Semantic Representation Component

The SOFIE Semantic Representation (SR) component enables semantic interoperability between applications and IoT systems by allowing the definition and enforcement of the semantics of data. The SR component offers two functionalities

- Data semantic definition
- Data validation

As shown in Figure 6.1, systems implementing the SR component enable interoperability by defining a data semantic which can be used by other entities to exchange data. The interoperability is achieved with a data model, defined using JSON schemas and W3C WoT Thing Description (WoT-TD)[3].

This data model is then managed by the SR component, which allows users to define the accepted data models in the system. The benefits of using Semantic Representation is to make systems more transparent to external users.

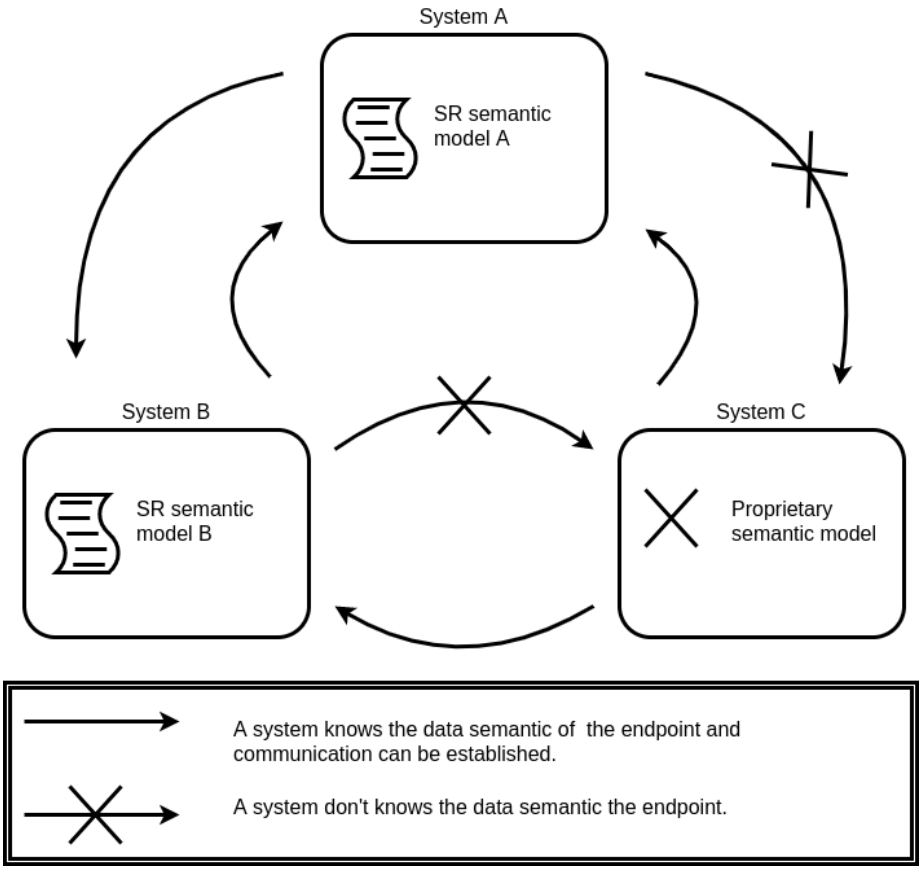

Figure 6.1: Shared data models enable interoperability.

The second benefit is message validation, which assures the quality of the data exchanged between systems. This is achieved by validating the messages external systems send to the SR component. The functionality is implemented by validating the messages against the schema defined with the data definition functionality. An example is JSON message validation:

[3] https://www.w3.org/TR/wot-thing-description/

this functionality acts as a filter for the data exchange to the system, informing the users whether the process is successful or what problems occurred.

The use of WoT Thing Description provides definitions that are broadly applicable to almost all IoT devices, such as basic descriptive information, security domain information, and details about the properties the device makes externally visible (such as temperature or humidity), about the supported actions (displaying an alert, for example), and events that the device generates that can be externally observed (e.g. a temperature threshold alert).

The WoT TD also natively supports multi-domain ontologies, as it supports the JSON-LD (JSON linked document) standard, allowing simultaneous use of different globally defined or locally (across participating organizations) ontologies. This allows, for example, a device's WoT-TD to be extended by a new linked schema, without compromising the interoperability with services that understand only the basic WoT-TD schema. This enables companies to quickly adapt to changes in the requirements by adapting custom schemas, enabling point-to-point interoperation between organizations, while allowing the schema to be later updated by a more formal standard.

The SR component can be used with other components of the SOFIE Framework to enhance the functionalities of a system. An example is the P&D component that uses the SR component to enforce rules and requirements for the provisioning of devices.

6.1 Requirements and Validation

Table 6.1 summarises the requirements for the Semantic Representation component from SOFIE deliverable D2.6.

Table 6.1. Requirements related to the Semantic Representation component.

Req. ID	Requirement Description	Priority	Category	Source
RF15	SOFIE must define an IoT things description model based on well-known standards (e.g. W3C standards).	MUST	AUDITABILITY	FSC, CAMG, S
RF16	SOFIE must implement standardised metadata and data representation formats and support various data modalities.	MUST	AUDITABILITY	FSC, CAMG, S
RF17	The semantic representation model of the system must be open and extensible by third parties (e.g. support the extension of the existing knowledge base and associations by extracting supplementary triples from RDF documents).	MUST	AUDITABILITY	FSC, CAMG, S
RF18	SOFIE must provide service discovery and resources selection processes based on multiple criteria over the features, associations, and interaction patterns of integrated resources.	MUST	INTEROPERA-BILITY	CAMG, S

RF19	SOFIE should support the semantic update and enhancement of resources' descriptions and associations in a dynamic way.	SHOULD	INTEROPERA-BILITY	CAMG

Table 6.2 details how the component has been validated with the specific tests. All tests are available in the component repository at the specified location and all tests have successfully passed, and with them the component meets all the requirements.

Table 6.2 Validation of the Semantic Representation component.

Semantic Representation		
RF15	*Requirement Description*	SOFIE must define an IoT things description model based on well-known standards (e.g. W3C standards).
	Test approach	Functional test
	Test Description	The test shows that only objects conforming to the component schema (W3C standards) are validated.
	Test location	Semantic Representation: tests/test_api.py -> test_api_validate()
RF16	*Requirement Description*	SOFIE must implement standardised metadata and data representation formats and support various data modalities.
	Test approach	Documentation
	Test Description	The component uses JSON objects.
	Test location	Semantic representation's repository documentation, "Main decision" chapter
RF17	*Requirement Description*	The semantic representation model of the system must be open and extensible by third parties (e.g. support the extension of the existing knowledge base and associations by extracting supplementary triples from RDF documents).
	Test approach	Functional test
	Test Description	The test shows how is possible to add a schema and subsequently add a schema extension. A message then is validated against both the extended schema and the schema extension.
	Test location	Semantic Representation: tests/test_api.py -> test_api_extended_validation()
RF18	*Requirement Description*	SOFIE must provide service discovery and resources selection processes based on multiple criteria over the features, associations, and interaction patterns of integrated resources.
	Test approach	Documentation
	Test Description	The SMAUG architecture shows how it is possible to provide service discovery and resource selection combining semantic representation and Discovery & Provisioning components
	Test location	https://github.com/SOFIE-project/SMAUG-Deployment
RF19	*Requirement Description*	SOFIE should support the semantic update and enhancement of resources' descriptions and associations in a dynamic way.

	Test approach	Functional tests
	Test Description	The test shows that a schema can be updated and enhanced with improved semantics.
	Test location	Semantic Representation: tests/test_api.py -> test_api_update_schema()

The SR component has reached TRL 7: it is an integral part of three SOFIE pilots, including the TRL 7 pilots Food Supply Chain and Decentralised Energy Flexibility Marketplace as detailed in SOFIE Deliverable D5.4 [D5.4].

6.2 Services and Interfaces

The Semantic Representation component provides the following 2 interfaces:

- **CRUD Data Schema:** These are REST APIs to manage the Data Schema. With these APIs it is possible to create, read, update and delete the JSON Data Schema.
- **Message Validation:** This REST API allows external actors to validate JSON messages with the SR's Data Schema.

The interfaces are depicted in Figure 6.2 and detailed in Table 6.3.

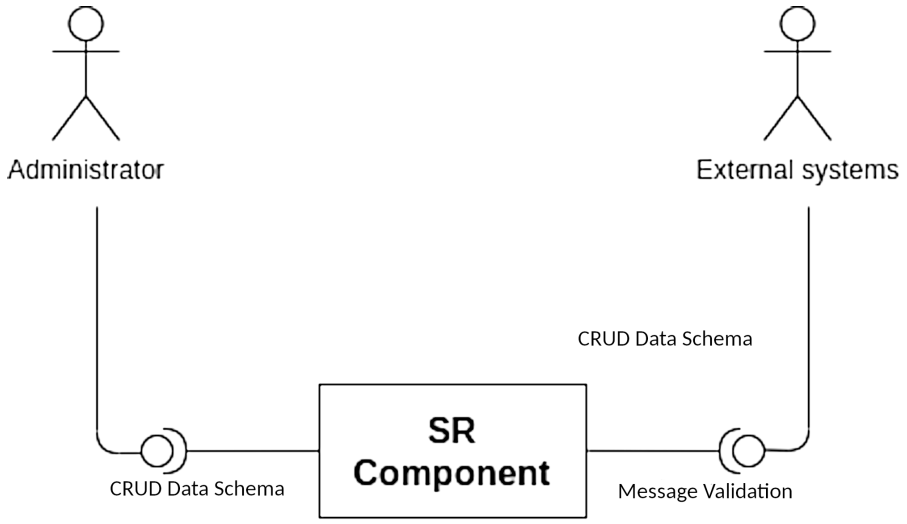

Figure 6.2: Semantic Representation Interfaces.

Table 6.3. Interfaces of the component

ID	Interface Content	
IF01	Name	**CRUD WoT TD Schema**
	Description	This collection of interfaces allow the component owner to manage JSON schemas in the components
	Key inputs	JSON schema
	Response	No output
IF02	Name	**Message validation**
	Description	This interface allow the validation of a JSON message
	Key inputs	JSON message

	Response	Valid message / which the message is not valid

6.3 The internal structure

The SR component consists of the following parts as shown in Figure 6.3:

- W3C IoT Thing Description Validator, which validates incoming JSON messages against JSON schemas saved in the DB.
- SOFIE Thing Description Schema, which manages the JSON schemas in the DB.

The Description validator is a JSON validator used to validate external parties' incoming JSON messages as shown in Figure 6.4. When an external party sends messages to a SOFIE system, the Semantic Representation, with its Description validator component, checks that the JSON structure is correct, then it checks that the incoming message is compliant with the rules defined in the JSON schema. The JSON schema is managed with the Description Schema component. When an external client sends a valid message to the SOFIE system, the semantic representation component will inform the client that the message is validated. When messages do not pass validation, the client receives information about what was not correct with the message.

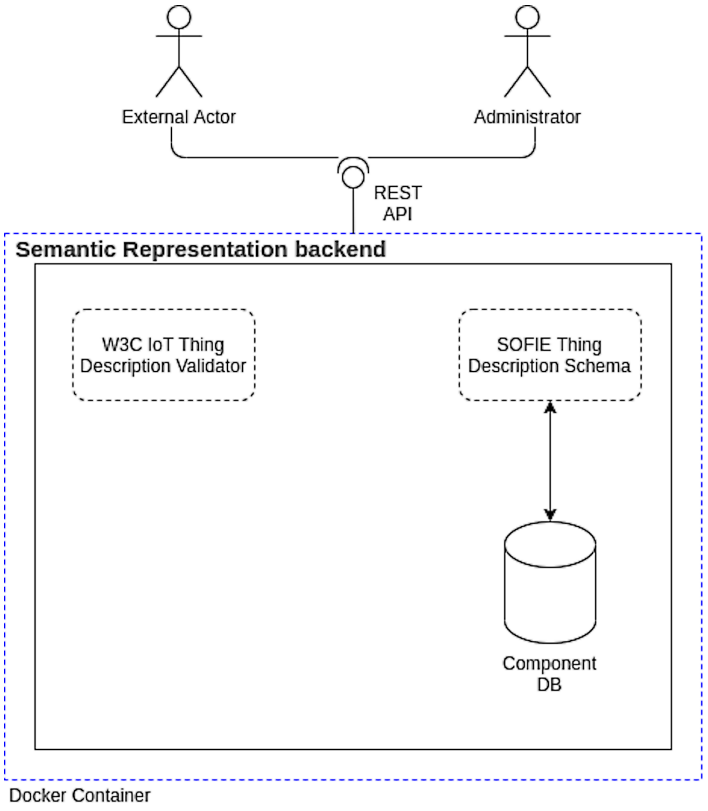

Figure 6.3: Semantic Representation internal structure

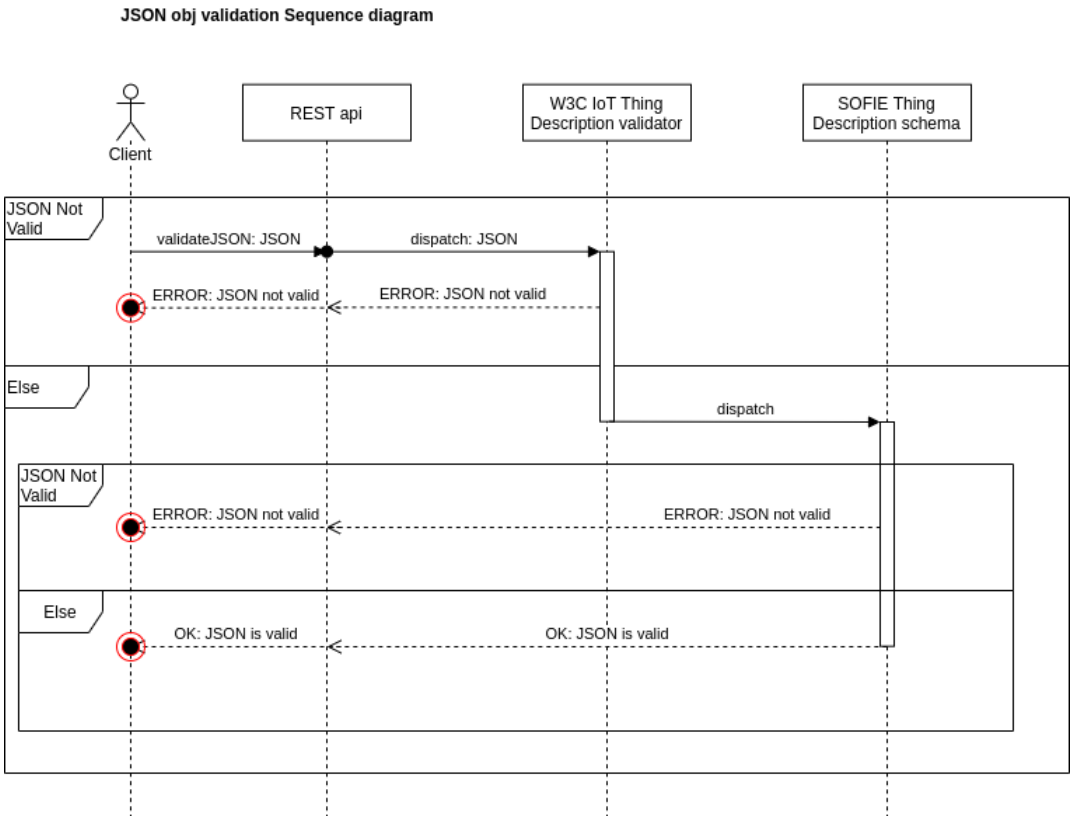

Figure 6.4: Message Validation sequence diagram.

7 Marketplace Component

The SOFIE Marketplace component enables the trade of different types of assets (e.g. electricity for charging a vehicle) in an automated, decentralised, and flexible way. A decentralised marketplace has the capability of operating without a single entity owning or managing it, which in turn increases competition and enhances its security, resiliency, transparency, and traceability. The decentralised marketplace can be either partially decentralised, when, e.g. a group of independent agriculture producers and retailers are managing it, or fully decentralised, when anyone can join and use the marketplace.

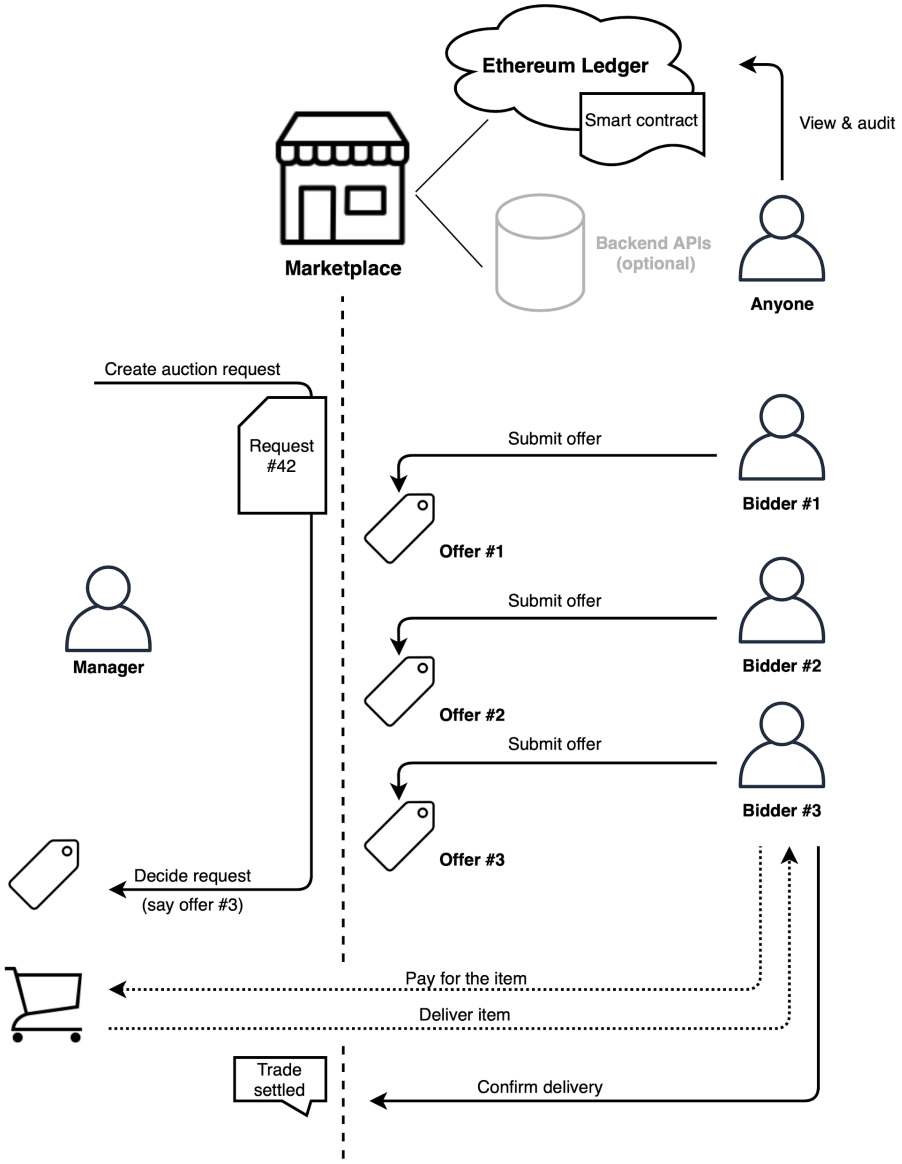

Figure 7.1: The data flow of an auction trade enabled by the Marketplace component.

Figure 7.1 shows the key functions of the Marketplace: the Manager can create auctions, Bidders can make bids for the item, after which the Manager decides the winner based on the type of the auction. Once the winner has paid and the item has been delivered, the winner can then confirm the receipt, thus concluding the transaction.

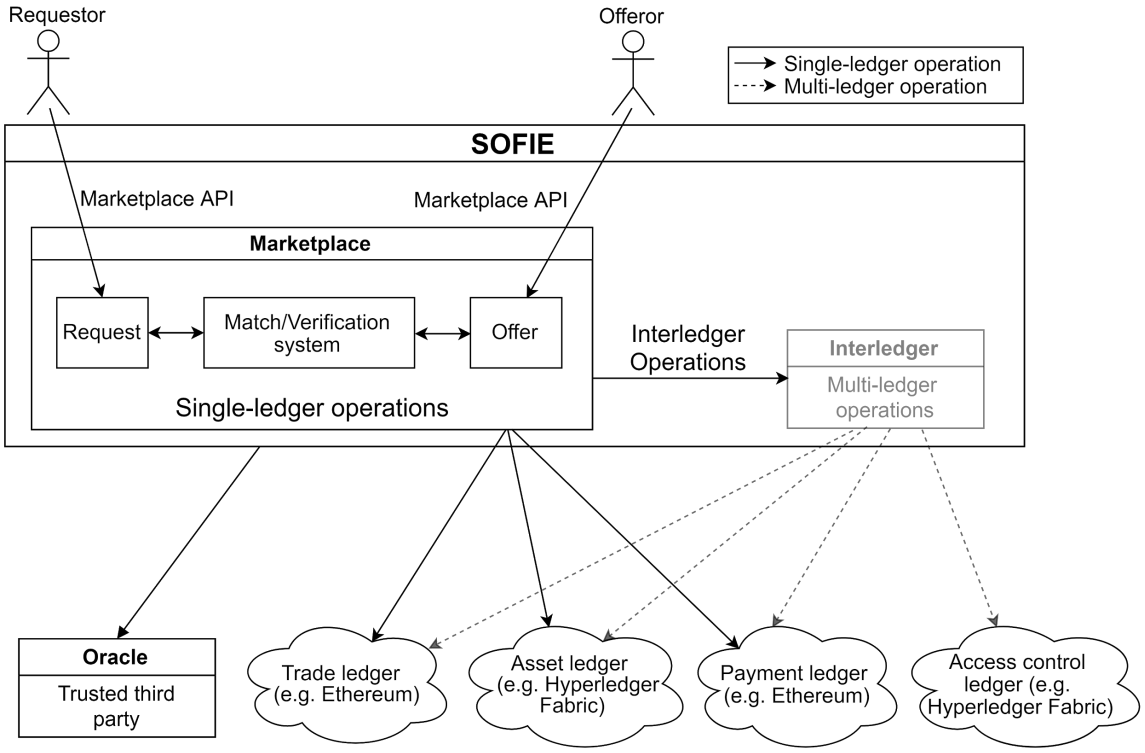

Figure 7.2: The marketplace component interacts with other components and ledgers.

Figure 7.2 shows a few examples of the marketplace component interacting with other components and ledgers. The marketplace interacts with the ledgers directly, unless it has to perform atomic operations involving multiple ledgers, in which case it uses the Interledger component. For instance, in the location based game pilot the marketplace component directly interacts with both the trade and asset ledgers to get information, and uses the interledger component to execute the trade. In the energy flexibility pilot the marketplace component again interacts with the asset and payment ledgers, and relies on an oracle to report on the charging activity as explained in Section 3.

7.1 Requirements and Validation

Table 7.1 summarises the requirements for this component (from SOFIE deliverable D2.6).

Table 7.1. Requirements for the Marketplace component.

Req. ID	Requirement Description	Priority	Category	Source
RF20	The marketplace must log the configuration of all trading actions (including offers, bids, parameters of resources, transactions etc.).	MUST	QUALITY	DEFM, CAMG, S
RF21	The marketplace must provide actors the capability to post/claim offers and sell/negotiate/exchange/buy resources and digital objects.	MUST	INTEROPERA-BILITY	DEFM, CAMG, S

RF22	The marketplace must support transparent trading of resources, i.e. the bids/offers matching process and the payments must be transparent.	MUST	OPERATIONAL	DEFM, CAMG, S
RF23	The marketplace must provide evidence once trades have been completed and resources have been properly delivered to the buyers.	MUST	SECURITY	DEFM, CAMG, S
RF24	The marketplace should allow integration of payment technologies.	SHOULD	OPERATIONAL	DEFM, CAMG, S

Table 7.2 details how the component has been validated with the specific tests. All tests are available in the component repository at the specified location and all tests have successfully passed, and with them the component meets all the requirements.

Table 7.2. Validation of SOFIE Marketplace component

ID	Validation Process	
RF20	*Requirement Description*	The marketplace must log the configuration of all trading actions (including offers, bids, parameters of resources, transactions etc.).
	Test approach	Functional test
	Test Description	The test sets up an auction, accepts bids, and decides which offer wins - and verifies all the related information is stored on the ledger.
	Test location	Marketplace: solidity/test/flowermarketplace
RF21	*Requirement Description*	The marketplace must provide actors the capability to post/claim offers and sell/negotiate/exchange/buy resources and digital objects.
	Test approach	Unit tests
	Test Description	The test sets up an auction, accepts bids, and decides which offer wins (and verifies all the related information is stored on the ledger).
	Test location	Marketplace: solidity/test/flowermarketplace
RF22	*Requirement Description*	The marketplace must support transparent trading of resources, i.e. the bids/offers matching process and the payments must be transparent.
	Test approach	Functional test
	Test Description	The test sets up an auction, accepts bids, and decides which offer wins - and verifies all the related information is stored on the ledger.
	Test location	Marketplace: solidity/test/flowermarketplace
RF23	*Requirement Description*	The marketplace must provide evidence once trades have been completed and resources have been properly delivered to the buyers.
	Test approach	Functional test
	Test Description	The transaction determining the winning bid is logged on the distributed ledger. Evidence of the delivery of resources must also be logged on the

		distributed ledger by the winner and seller, after which the evidence can be verified.
	Test location	Marketplace : solidity/contracts/interfaces/TradeResource.sol
RF24	*Requirement Description*	The marketplace should allow integration of payment technologies.
	Test approach	Documentation
	Test Description	The marketplace component provides interfaces for integrating payment solutions and an example from the Energy Flexibility pilot provided by Engineering integrates the ERC20 tokens payment in the energy marketplace.
	Test location	Marketplace: solidity/vendors/ENG/EnergyMarketPlace.sol

The MP component has reached TRL 7: it is an integral part of two SOFIE pilots, including the TRL 7 pilot Decentralised Energy Flexibility Marketplace as detailed in SOFIE Deliverable D5.4 [D5.4].

7.2 Services and Interfaces

Figure 7.3 presents a marketplace that communicates with smart contracts and stores the data both in the database and on the blockchain. Customers can use this marketplace with a known URL or with a web application.

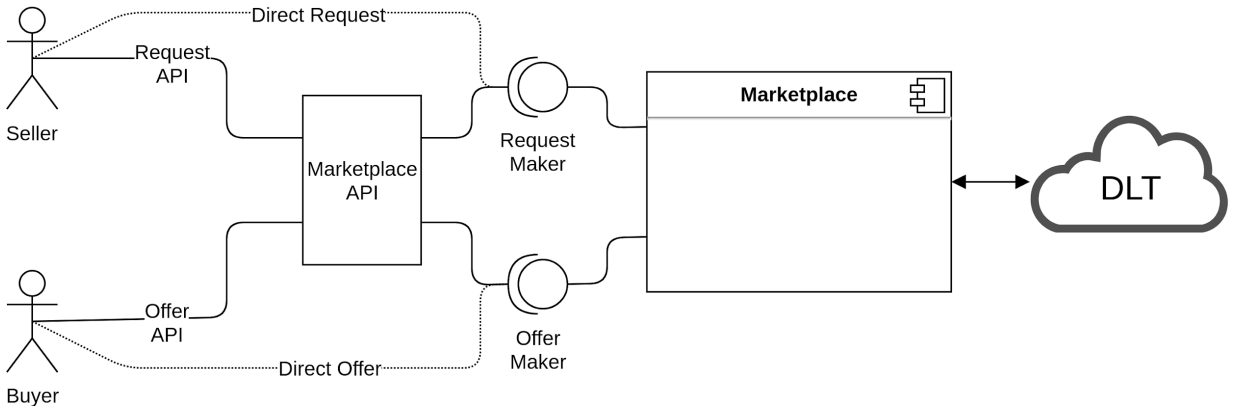

Figure 7.3: The Marketplace component and its interfaces.

The Marketplace component offers three interfaces: Request Maker for sellers to create, manage and conclude auctions, Offer Maker for buyers to participate and bid in auctions, and Event callbacks that provide a mechanism for monitoring events on the Marketplace. The interfaces are described in Table 7.3.

Table 7.3. Interfaces of the component

ID	Interface Content	
IF01	Name	**Request maker**

	Description	This interface has several functions for creating and submitting requests (to sell assets), for the matching process (of requests and offers), and for getting information about a request and its offers.
	Key inputs	Request ID
	Response	It creates requests, returns their information, matches them with offers automatically, and logs the actions.
IF02	Name	**Offer maker**
	Description	This interface has several functions for creating and submitting offers (to buy assets) and getting information about an offer.
	Key inputs	Request ID, Offer ID
	Response	It creates offers, returns their information, and logs the actions.
IF03	Name	**Event callbacks**
	Description	This interface allows the registration of callbacks that will be called after the Marketplace smart contract emits some event
	Key inputs	Event type, callback endpoint
	Response	The callbacks endpoints will be invoked by the corresponding events emitted.

7.3 The internal structure

The marketplace is used for trading assets (e.g. energy, digital assets, resource access, etc.). Figure 7.4 presents the internal structure of the marketplace component. The Marketplace module includes functionality for communicating with marketplace smart contracts (which are shown with dotted lines). The Marketplace interface smart contract includes offer maker and request maker interfaces. The Marketplace base includes all of base functionalities for the marketplace component, while Ethereum standards includes the standard Ethereum tokens like ERC20[4].

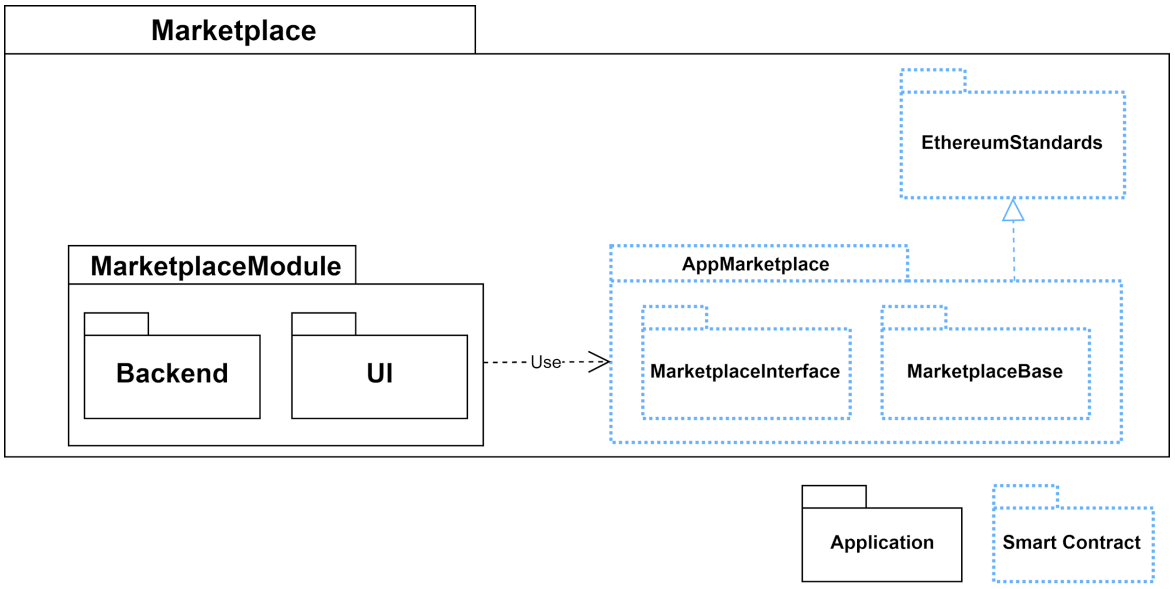

Figure 7.4: The package class diagram of the marketplace component

[4] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

The Marketplace component offers three key functionalities: Request creation, Offer creation, and Decision.

Request creation

For trading assets, request creation, depicted in Figure 7.5, is the first step. The manager of an asset creates a request to sell the asset, and customers can then make offers based on the request.

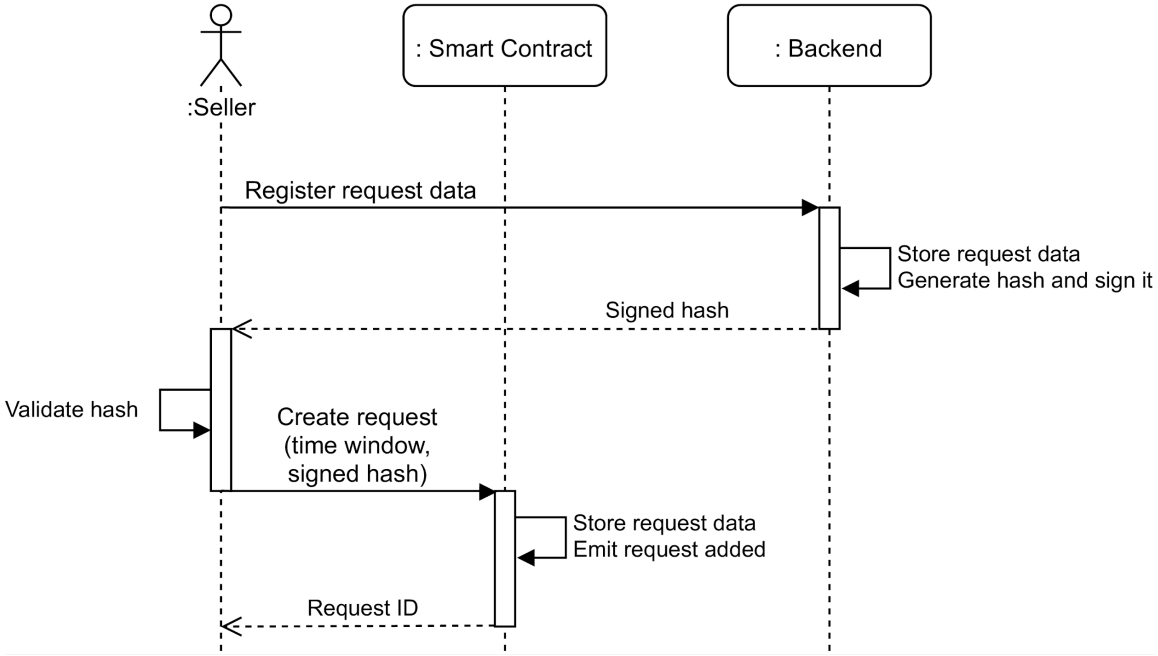

Figure 7.5: Request creation sequence diagram

Offer creation

Once a request has been created, customers can then submit offers on the request. Figure 7.6 shows how an offer can be created.

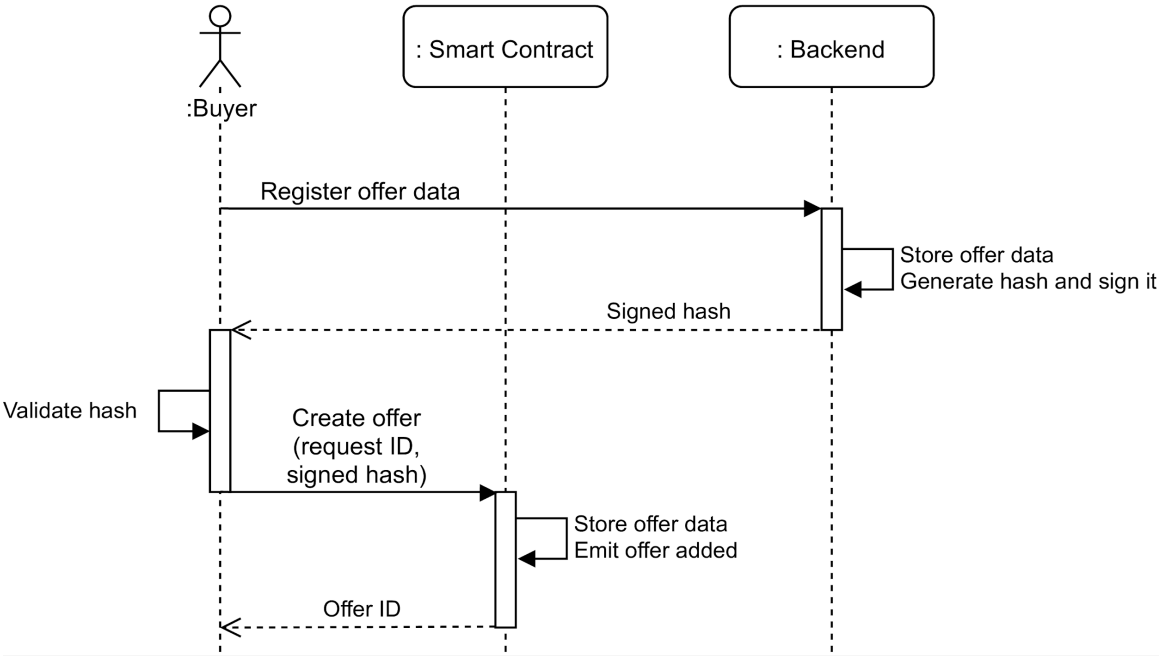

Figure 7.6: Offer creation sequence diagram.

Decision process

When the deadline for offers has passed or the requestor wants to determine the result, the decision process should be run to choose the winning offer. Figure 7.7 shows how the decision process happens.

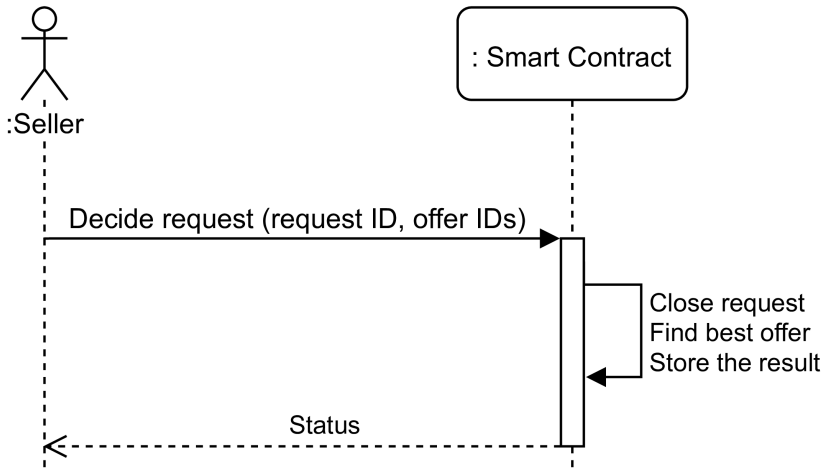

Figure 7.7: Decision process sequence diagram.

The current version of the smart contract implements an auction mechanism, in which the best offer is selected following the “lowest bidder” rule. In the future, the smart contract may be upgraded to consider alternative ways to select the winning offer, e.g. in addition to the price also taking other features of the offer into account. In the case of balancing the load on the electricity grid, these can include, e.g. more fine-grained promises to use energy during a specific time.

Proof of resource delivery

The TradeResource interface allows the winner of an auction to confirm that the traded item has actually been delivered, and thus, the auction as a whole has been settled. The interface has been implemented as the *settleTrade(requestID, offerIDs)* in the *Abstract Marketplace* interface, where an event of *TradeSettled(requestID, offerIDs)* is emitted, so that applications can subscribe and get notified once the resource is delivered. Applications can further expand the logic in the smart contracts to allow e.g. external oracle addresses to confirm the delivery of the item.

8 Provisioning and Discovery Component

The goal of the Provisioning & Discovery (P&D) component is to manage the IoT resources in the system by provisioning the existing IoT devices to a working state and by enabling the discovery of new IoT resources along with their metadata. Using this functionality, it is possible to e.g. decentralise the process of making new resources available to systems and to automate the negotiations for the terms of use and the compensation for the use of these resources. This component works together with the SOFIE Semantic Representation component to provide meta-data for the IoT devices.

The SOFIE Framework's P&D component provides the following functionalities:

- Provisioning of IoT resources including configuration of devices and enrolling new devices to system
- Discovery of the new IoT resources using e.g. Bluetooth Low Energy (BLE) Discovery and DNS-Service Discovery
- Licensing of the resources

The first functionality of the component is to provision the devices using the meta-data provided by the SOFIE semantic representation file. The process of provisioning involves enrolling a device into the system and getting each device configured to provide the required service. In the component, the Provisioning interface goes through the meta-data and checks against the requirement before provisioning the device to the database. This also acts as a filter for either accepting or rejecting the newly discovered IoT resource. After enrolling the device, the provisioning interface provides the configuration for the device to bring it to a working state with the deployed platform.

The second functionality of the component is the discovery of new IoT resources. The Bluetooth discovery interface provides operations to perform a BLE scan to discover open IoT devices nearby. It also provides a LAN discovery interface to discover devices published on the local (WLAN, etc.) network. The interfaces list newly discovered devices along with their meta-data before enrolling them in the system.

The final functionality of the component is to license the device to automate the negotiations for the terms of use. The interface then later calls a smart contract on the blockchain and compensates the owner of the device for the usage of the provisioned device.

8.1 Requirements and Validation

Table 8.1 summarises the requirements for the provisioning and discovery component introduced in this document.

Table 8.1. Requirements related to the SOFIE provisioning and discovery component

Req. ID	Requirement Description	Priority	Category	Source
RF18	SOFIE must provide service discovery and resources selection processes based on multiple criteria over the features, associations, and interaction patterns of integrated resources.	MUST	INTEROPERA-BILITY	CAMG, S

RF28	The component must provide actors the capability to configure the discovered IoT device	MUST	OPERATIONAL	CAMG, S
RF29	The component must provide actors to provision the device to the database.	MUST	OPERATIONAL	CAMG, S
RF30	The component should be able to discover new IoT devices using multiple protocols	OPTIONAL	INTEROPER-ABILITY	CAMG, S

Table 8.2 details how the component has been validated with the specific tests. All tests are available in the component repository at the specified location and all tests have successfully passed, and with them the component meets all the requirements.

Table 8.2.Validation of the SOFIE provisioning and discovery component

ID	Validation Process	
RF18	*Description*	The component should be able to start discovery service on the IoT device.
	Test approach	Functional test
	Test Description	The test sets up an raspberry pi to act as an IoT device and enable discovery services.
	Test location	Provisioning and discovery: /tests/tests_run.py
RF28	*Description*	The component should be able to configure the IoT device to work with the IoT platform.
	Test approach	Functional test
	Test Description	The test configure the raspberry pi to work as an IoT beacon
	Test location	Provisioning and discovery: /tests/tests_run.py
RF29	*Description*	The component should be able to provision the discovered device to the database
	Test approach	Documentation
	Test Description	The mobile client provision the device to the given URL database
	Test location	Provisioning and discovery: Android Application
RF30	*Description*	The component should be able to discovery IoT devices using multiple protocols
	Test approach	Functional Test
	Test Description	The test sets up the raspberry pi to be discovered over bluetooth and DNS.
	Test location	Provisioning and discovery: /tests/tests_run.py

The P&D component has reached TRL 6: it is an integral part of the TRL 6 SOFIE pilot Context-Aware Mobile Gaming as detailed in SOFIE Deliverable D5.4 [D5.4].

8.2 Services and Interfaces

The four interfaces of the component have been detailed in Table 8.3: there are separate interfaces for discovery via Bluetooth and LAN, an interface for provisioning and a licensing interface.

Table 8.3. Interfaces of the component.

ID	Interface Content	
IF01	Name	**Bluetooth discovery**
	Description	This interface provides operations to perform a Bluetooth scan and discover open Bluetooth devices
	Key inputs	Bluetooth scanning interval and timeout.
	Response	It lists all the Bluetooth devices and downloads the device description file using the URL provided by the Bluetooth service.
IF02	Name	**Local network discovery**
	Description	This interface provides operations to perform a Multicast-DNS scan to find beacon services published on the local (WLAN, etc.) network.
	Key inputs	Service type.
	Response	It lists all the devices using the webthing service and downloads the device description file using the URL provided by mDNS discovery.
IF03	Name	**Device provisioning**
	Description	This interface goes through the semantic representations and checks them against the requirement for provisioning of the device.
	Key inputs	Semantics representation file of discovered beacons.
	Response	It either adds the device to the client database or rejects it.
IF04	Name	**Licensing provisioning**
	Description	The interface checks for the license in the semantic representation file and automatically makes a contract for usage of the device.
	Key inputs	Semantics representation file and information for device usage.
	Response	It deploys a contract and compensates for the usage of the provisioned device.

Figures 8.1 and 8.2 present the information flow in the discovery and registration process. Here, the resource types, provided services, licences and compensation are all described using WoT TD semantic representations.

In Figure 8.1, the mobile application connects to the local network and starts searching for the configured type of service on the network. After getting the list of matching services, the application requests for the semantic descriptions of the devices and checks that they meet the specified requirements. Finally, it provisions the accepted devices' information to the database and updates the device configuration to work with the platform.

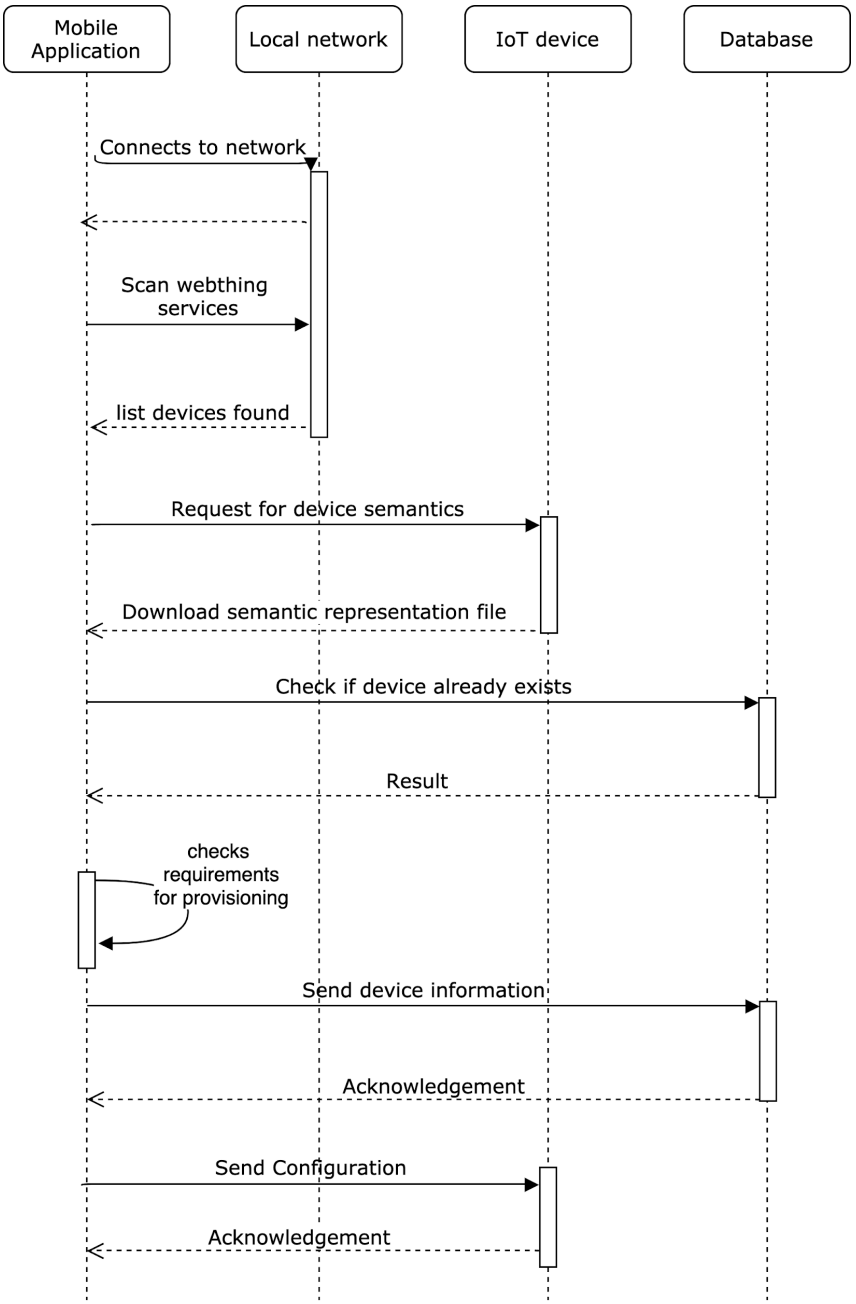

Figure 8.1: The process of discovering new devices on a local network.

In Figure 8.2, the user selects the bluetooth protocols from the mobile client, which starts scanning for nearby BLE devices and shows the list to the user. The mobile client then downloads the semantic representation file of the devices and checks it against the requirements. Finally, it provisions the accepted devices and their metadata to the database and connects the device over bluetooth to send the configuration details.

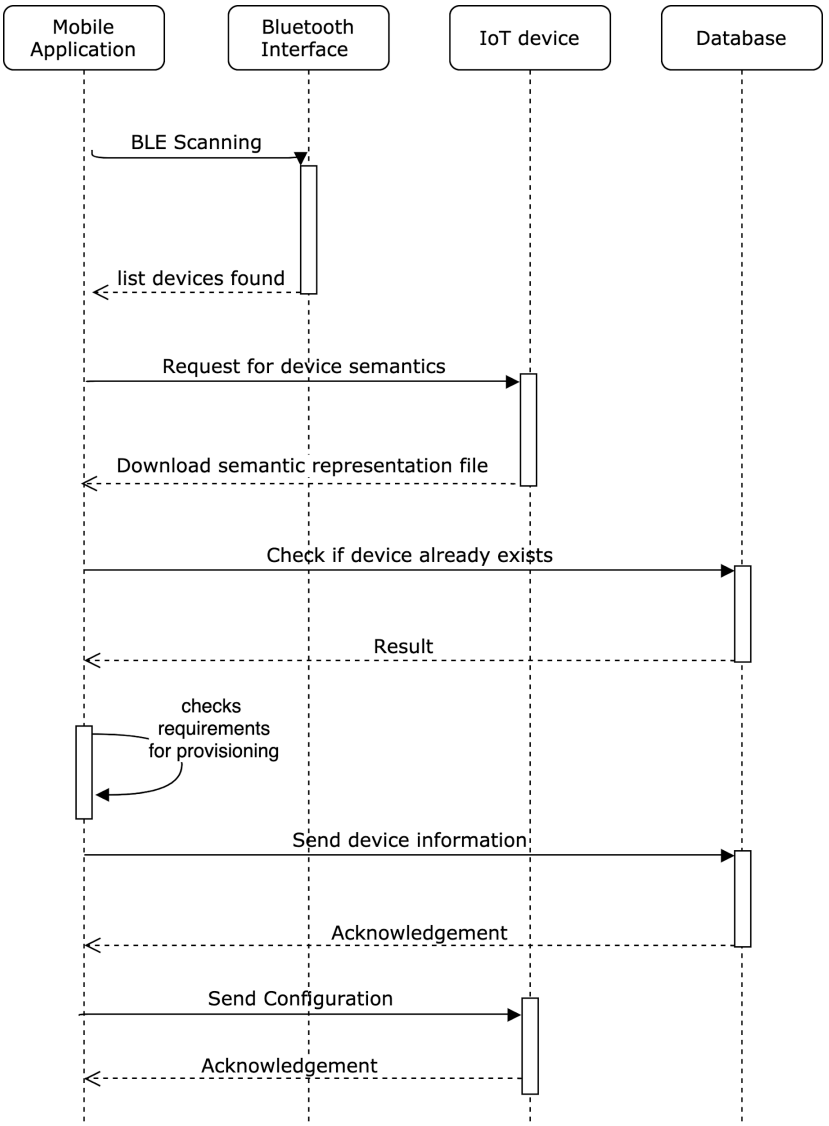

Figure 8.2: The process of discovering new devices using BLE.

8.3 The internal structure

The structure of the Provisioning and Discovery component is shown in Figure 8.3. There are separate functionalities for Bluetooth and LAN discovery, and an example mobile app for performing device discovery.

Bluetooth Low Energy (BLE)
Bluetooth Low Energy is a form of wireless communication designed especially for short-range communication. In this component, a Bluetooth device with a custom advertising packet is used to discover the IoT devices in proximity and configure them to be used as beacons. The Bluetooth GATT (Generic Attribute Profile) is the foundation for the design of any BLE system and defines the way an application interacts with the end-device. It is used after a connection has been established between the two devices. When a BLE device is advertising it will periodically transmit packets containing device information. In this component, a custom advertising packet contains the custom name, service UUIDs, and custom data i.e. URL. We use “0x24” URI for Advertising Data type for encoding URLs. It is important to know that an advertising packet can consist of no more than 31 bytes.

Eddystone is an open beacon format developed by Google and designed with transparency and robustness in mind. Eddystone can be detected by both Android and iOS devices. The Eddystone-URL frame broadcasts a URL using a compressed encoding format in order to fit more within the limited advertisement packet. In this component, an Eddystone-URL is used to broadcast the URL of the Semantic Representation file, then any client that received this packet can download the meta-data of the specific device.

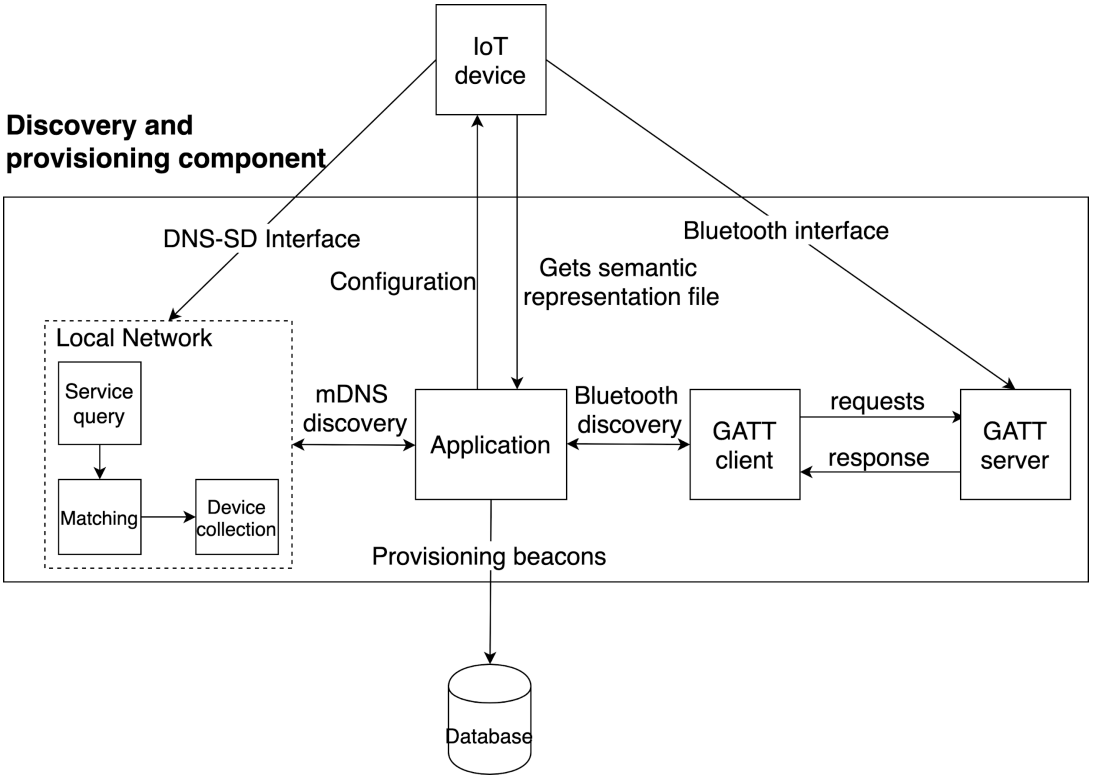

Figure 8.3 : *SOFIE Provisioning and discovery Internals.*

DNS Service Discovery with multicast (DNS-SD)

DNS service discovery (DNS-SD) allows clients to discover a named list of service instances of a given service type and to resolve those services to hostnames using standard DNS queries. It discovers devices and services on a local area network using IP protocols, without requiring the user to configure them manually. DNS service discovery requests can also be sent over a multicast link, and it can be combined with multicast-DNS (mDNS) to yield zero-configuration DNS-SD. DNS-SD can be used to discover the IoT devices available on the local (WLAN, etc.) network. It then provides access to the semantic file of the device, and the provisioning interface also uses the same link to configure the device.

Mobile Client

The mobile client was developed for Android mobile devices using Android Studio. The mobile client performs automatic background scans for discovering new IoT devices based on the selected protocol by the user. After discovering the device, the client downloads the semantic file and goes through the meta-data and checks it against the requirements. The mobile client then stores the device information to the database and sends configuration details to the device to work with the IoT platform.

9 Federation Adapters

Federation adapters (FAs) are used to link the IoT systems to the SOFIE Architecture. Due to the wide range of IoT systems SOFIE can be connected to, no single general-purpose adapter can be created. Rather, each IoT system requires a separate adapter that accomodate the individual requirements of the IoT system and the use case (including the business case). They can be implemented in two ways: a *thin* adapter only provides the required linking functionality and some SOFIE functionality with the rest of the SOFIE functionality being implemented separately, whereas a *thick* federation adapter includes all the utilised SOFIE components, making it a complete stand-alone system.

This section introduces the federation adapters from three SOFIE pilots: the Food Supply Chain (FSC) and Decentralised Energy Flexibility Marketplace (DEFM) pilots utilise the thin FA approach, while the Decentralised Energy Data Exchange (DEDE) pilot builds on a thick FA.

9.1 Requirements and Validation

Table 9.1 summarises the requirements for the federation adapters (from D2.6), and Table 9.2 then details, how all the FAs meet all the requirements.

Table 9.1: Requirements for the federation adapters.

Req. ID	Requirement Description	Priority	Category	Source
RF25	SOFIE deployments can utilise one or more Federation Adapters each capable of representing one or more IoT Devices/Platforms.	MUST	OPERATIONAL	FSC
RF26	The IoT device/platform must be able to utilise all the SOFIE functionalities it requires through the Federation Adapter representing it.	MUST	OPERATIONAL	FSC
RF27	Federation Adapters must not require changes to the IoT device/platform it represents.	MUST	OPERATIONAL	FSC, DEFM

Table 9.2: Validation of the federation adapters.

Req. ID	Requirement Description
RF25	• In the FSC pilot, Synfield, Aberon, and Transportation FAs are used to represent the Synfield, Aberon, and Transportation IoT platforms respectively • The DEFM pilot uses the DEFM and DEDE Federation Adapters. The first one is used to collect, store, and analyse data from the smart meters to identify the need for flexibility campaigns while the second one is demonstrated in the cross-pilot scenario. • In the DEDE pilot, the FA can represent one or more IoT Devices/Platforms by generating a separate DID for each one.
RF26	• In the FSC pilot, all IoT platforms (Synfield, Aberon, and Transportation) connect to the pilot's platform only via the SOFIE Federation Adapters (Synfield FA, Aberon FA, and Transportation FA)

	• In the DEFM pilot, the FA enables the IoT devices to provide the live & historical data used to create the Marketplace requests. • In the DEDE pilot, all the SOFIE functionalities are offered by the FA and thus are available to the IoT device/platform using it.
RF27	• In the FSC pilot, Synfield, Aberon, and Transportation FAs utilise the existing APIs provided by the Synfield, Aberon, and Kaa IoT platforms to federate them with the pilot's platform, without requiring any changes in the existing IoT platforms. • The DEFM pilot adapter leverages the FIWARE Context Broker and IoT Agents to enable the connection with IoT devices without any changes required. • In the DEDE pilot, the FA is a pre-packaged component that supports services described in the OpenAPI service description format. A converter can always be made to expose any existing service as an OpenAPI service.

The FAs have all reached TRL 7: they are integral parts of the TRL 7 SOFIE pilots Food Supply Chain, Decentralised Energy Flexibility Marketplace, or Decentralised Energy Data Exchange as detailed in SOFIE Deliverable D5.4 [D5.4].

9.2 Food Supply Chain

In the Food Supply Chain (FSC) pilot shown in Figure 9.1, three IoT platforms (Synfield, Aberon, and Transportation) are federated into the pilot platform. Each IoT platform is connected to the pilot platform through a federation adapter, which

- Provides an adaptation layer for data and resources to enable syntactic and semantic interoperability as well as secure usage of platform by exposing a RESTful API to the Supervisor Web Server (SWS).
- Implements a domain-specific API to communicate with the SWS and allows rapid cross-platform access and application development.
- Creates wallet addresses to register the IoT platform in the consortium ledger and digitally signs the data objects which are sent to the SWS.

Functionality from three SOFIE components (IAA, PDS, and SR) is included in the Food Supply Chain pilot Federation Adapters. The steps of the interaction between the SWS and the PDS and IAA components are the following:

- The FA adapter uses its Ethereum wallet address to construct a Decentralised Identifier (DID).
- Upon platform registration, the SWS configures the resource server (PDS) with the DID of the IoT platform.
- The FA of the IoT platform completes the challenge/response via an endpoint and it receives a JWT token.
- When the SWS interacts with the FA, the FA attaches an HTTP header, containing the JWT token along with its payload.
- The SWS invokes the authorisation component (IAA) to verify the JWT token against the given FA wallet address.
- If the response is HTTP_200 and the wallet address (embedded in step 2 in the sub field) matches the wallet address retrieved by Ethereum, the Federation Adapter payload is considered valid and secure.

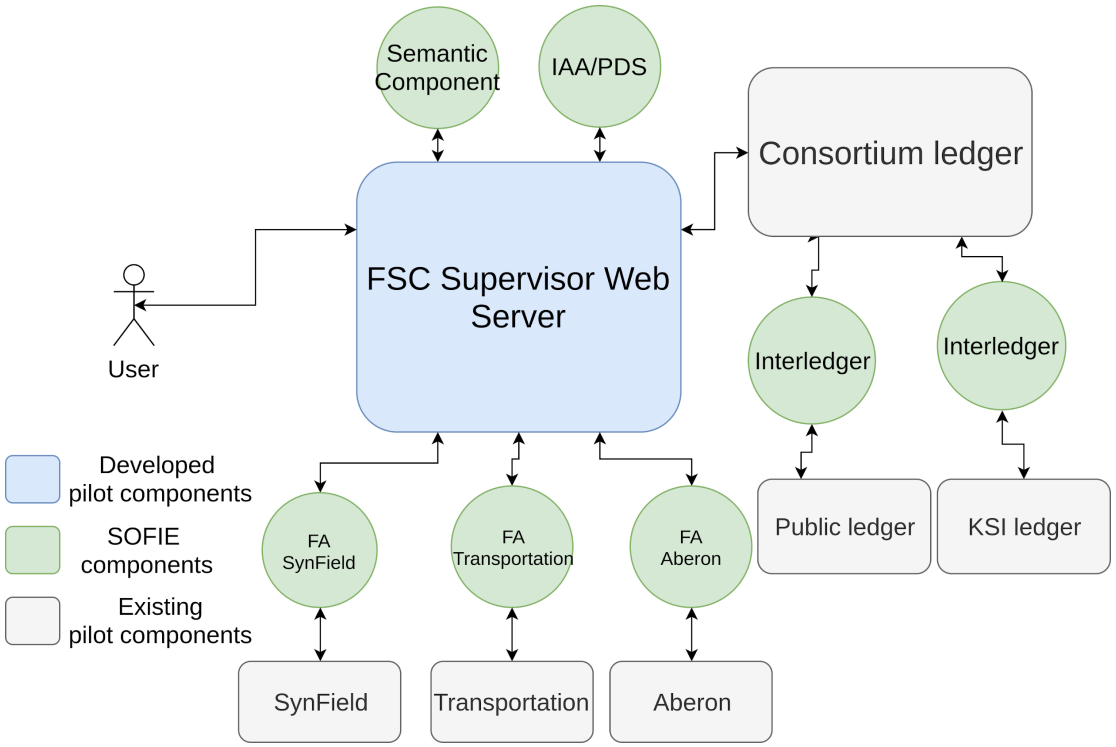

Figure 9.1: The Food Supply Chain pilot utilises three federation adapters

The Semantic Representation (SR) component is used in the Federation Adapter (FA) – Supervisor Web Server (SWS) communication. The FA has to provide an endpoint (usually the base endpoint '/' of the FA API) that, once accessed via an HTTP GET call, outputs the schema of the Federation Adapter. The SWS then reads the schema, retrieving information about the endpoints and functionality provided by the FA on behalf of the underlying IoT Platform. The flow can be described as follows:

- When a new IoT platform is registered via the web application by the pilot admin and the Ethereum transaction is mined, an event is fired. When the SWS acknowledges the event, the SR component is invoked on an endpoint and the FA schema is registered with the SR component.
- The SWS then retrieves the schema of the given platform, and via an HTTP POST request to the SR component to an endpoint, the platform is added.
- Upon every SWS communication with the FA, the FA payload is validated by the SR component against the given schema.
- If an IoT platform is removed from the FSC pilot platform, an Ethereum event is again generated. The SWS acts on this event to inform the SR component about the platform's deregistration by invoking the deregistration endpoint of the SR component.

More information about the adapters can be found in SOFIE Deliverable D3.5 [D3.5]. The FA for the Transportation IoT platform is available on GitHub along with installation and deployment instructions [FAs].

9.3 Decentralised Energy Flexibility Marketplace

The Federation Adapter (FA) used in the Decentralised Energy Flexibility Marketplace (DEFM) pilot enables the collection, storage, and analysis of the data gathered from the IoT smart meters. The FA is based on the FIWARE platform and consists of a *Context Broker*, an *IoT Agent*, a *Short-Term Historical* component, *MongoDB* no-sql database, *Mosquitto* MQTT

broker, the appropriate configuration, documentation and examples. The FA enables the organisation of *sensors* and *actuators* in *service groups*, and the management of the whole lifecycle of context data. It has been released as open-source software in the GitHub [FAs].

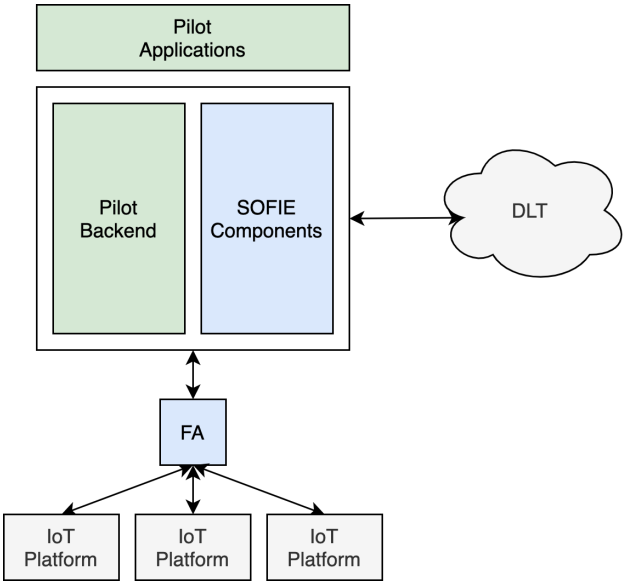

Figure 9.2: Relation of the DEFM FA with the other pilot components

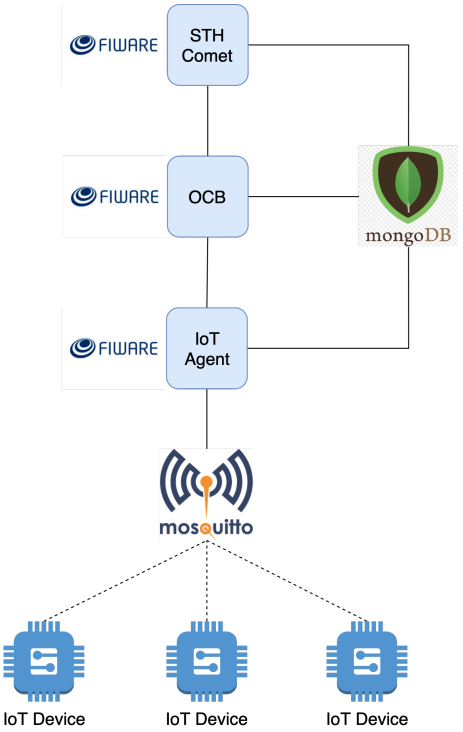

Figure 9.3: Detailed view of the DEFM FA architecture

The main function of the FA is the context management (i.e. information about entities) and the management of the context availability (i.e. information about the data providers). The relation between the FA and the other platform components in the DEFM pilot is represented in Figure 9.2, while Figure 9.3 details the individual components that constitute the FA.

9.4 Decentralised Energy Data Exchange

The Decentralised Energy Data Exchange (DEDE) pilot connects data providers with data consumers. Both the data providers and the data consumers connect to the platform through their own instance of the same Federation Adapter (FA), which is a common software component for both parties and needs no extension or customisation. All the SOFIE components utilised by the pilot have been integrated in the FA.

As shown in Figure 9.4, the FA of a data consumer connects directly to the FA of a data provider to exchange messages according to the DEDE FA communication protocol. Messages between the two FAs are transported securely over a mutually authenticated TLS connection, using Hyperledger Indy-based decentralised identifiers (DIDs) and verifiable credentials (VCs) to establish trust. The main function of the FA is to ensure interoperability and to secure the communication with other entities on the platform. It acts as a forward proxy for the data consumer and as a reverse proxy for the data provider, but it can also be in both

roles at the same time, enabling entities that are both data consumers and data providers. The FA takes care of the security aspects of the integration and lets the data provider concentrate on implementing services and the data consumer to use these services. The only requirement for a data provider is to describe its services according to the OpenAPI 3.0 specification.

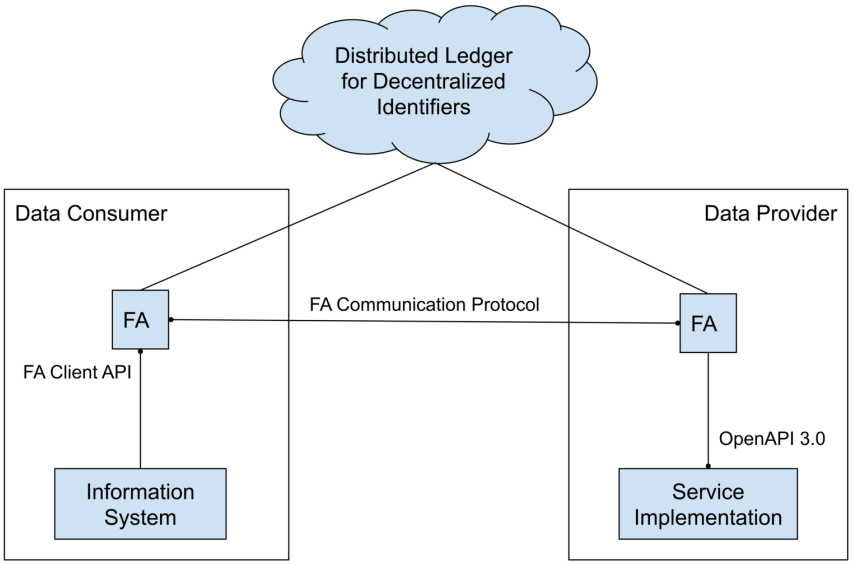

Figure 9.4. The role of the FA in the DEDE pilot.

From the implementation perspective, the two main responsibilities of the FA is to proxy messages and to manage the identity of the represented entity. The internal structure of the FA shown in Figure 9.5 mirrors this with two loosely coupled services: proxy and ssi-agent (Self-Sovereign Identity agent). Both of the components have public and private interfaces, for external and internal use, respectively.

The information system of the data consumer first sends a request to the proxy's private interface. The proxy uses the ssi-agent private interface to resolve the endpoint of the target DID and to sign the request with the source DID. Then, it initiates a secure connection to the public interface of the data provider (target DID) proxy. Both sides use the ssi-agent private interface to retrieve the hash of the currently valid certificate to verify the authenticity of the connection.

Once the connection is set up, the data provider proxy will use the ssi-agent private interface to verify the request's signature. If the request is for a service that requires further authorisation, the data provider proxy will also use the ssi-agent private interface to get the proved values of the attributes required for the authorisation decision. If the data provider ssi-agent receives such a request, it will send a proof request to the public interface of the data consumer ssi-agent. Once the data provider proxy has values for all the proved attributes, it can forward the request to the service implementation that is described using the OpenAPI 3.0 specification. The signing of the response and the verification of the response message signature is analogous to the processing of the request.

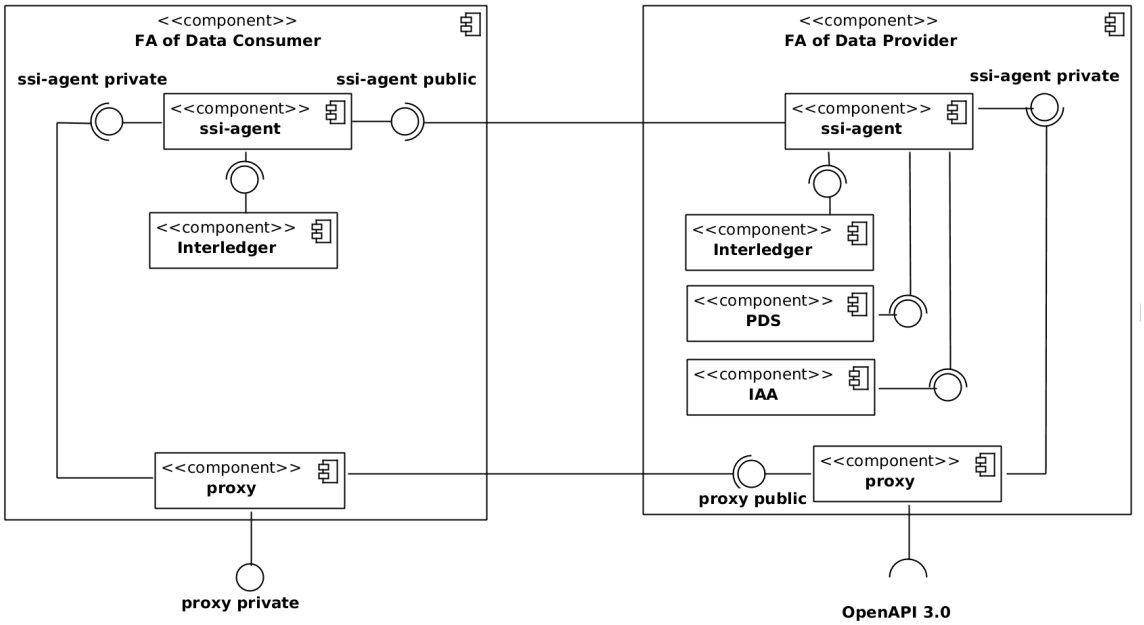

Figure 9.5. Internal structure of the DEDE FA.

10 How Components are used in the SOFIE Pilots

This section describes how the SOFIE pilots and the SMAUG reference application each utilise the Framework components to provide key benefits to the use cases. More details about the pilots and SMAUG can be found in D5.4 [D5.4].

The pilots utilise the SOFIE components as detailed in Table 10.1.

Table 10.1. SOFIE component usage in each pilot and reference application SMAUG.

Pilot	FCS	DEFM	CAMG	DEDE	SMAUG
Interledger	X	X	X	X	X
Identity, Authentication, and Authorisation	X			X	X
Privacy and Data Sovereignty	X			X	X
Semantic Representation	X	X	X		X
Marketplace		X	X		X
Provisioning and Discovery			X		X

10.1 Food Supply Chain

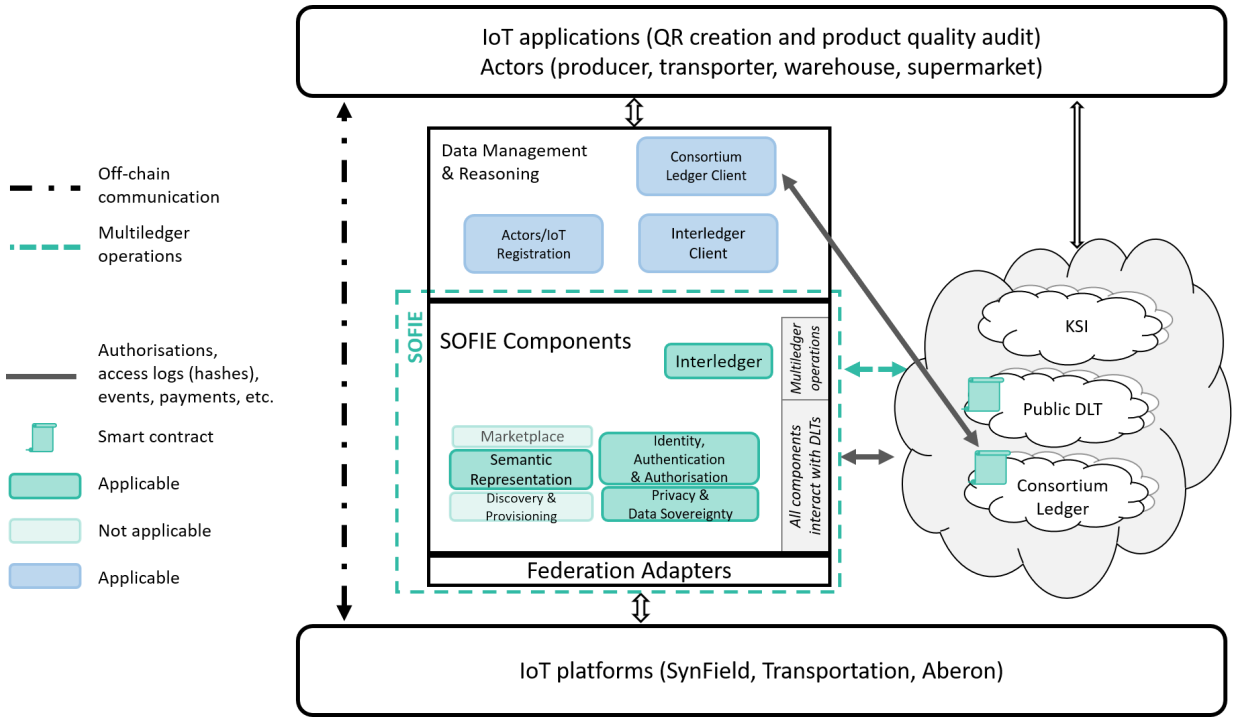

Figure 10.1: Food Supply Chain pilot platform architecture

In the Food Supply Chain (FSC) a *consortium distributed ledger* is leveraged to federate three different IoT platforms and to establish a distributed and immutable data management layer that provides traceability and quality control services for transported products. This makes

traversing the path from field to fork more robust, reliable, and time-efficient for all parties involved in the food supply chain. Figure 10.1 presents the pilot's architecture in relation to the utilised SOFIE framework components.Two main services (bundled in a web application) are being offered via the pilot's platform, namely the usage of QR codes to encode *product history* from the field to the market shelf, and product *quality audits and resolution of disputes* for product quality degradation events. To offer these services, several SOFIE framework components are utilised:

Interledger component

The FSC pilot uses the following ledgers:

1. A private consortium ledger (Ethereum) to store IoT data which is collected from the three companies operating the farming, transportation and warehouse segments, respectively. This data captures both the conditions and the handoffs of the trackable boxes as they move along the supply chain.
2. A public ledger (Ethereum ROPSTEN), where hashes from the above mentioned transactions in the consortium ledger are stored in a master hash. This master hash is used by external entities (i.e. entities that do not have direct access to the consortium ledger, such as the supermarket organisation and the customers) to verify the authenticity of data requested from the consortium ledger.
3. A KSI blockchain to create a unique signature (anchor) per master hash created in the public ledger. The KSI FSC Pilot integration happens at the final stage of the supply chain.

The interledger is being used in the following ways:

- for storing hashes: one Interledger instance is used to store the hashes to the public ledger and another instance is used with KSI.
- for a proof of integrity operation to cross check data on the consortium and public ledger against the signatures stored in the public ledger and the KSI, thus validating the integrity of the data in audits on behalf of external entities.

Thanks to the IL component, interoperability between DLTs is achieved, allowing to use a combination of private/permissioned and public/permissionless DLTs for enhancing trust and data integrity.

Identity, Authentication and Authorisation component

In the FSC pilot, two IAA mechanisms are implemented for the actors and the IoT platforms, respectively. More specifically:

1. Authentication and authorisation of actors: The username/password method of the IAA component is used to authenticate the actors of the various business segments. The AA API of the Oauth2 server (keycloak) is used by the FSC web application which implements the OAuth2 client. The Oauth2 server provides the client with authorisation tokens, so actors are able to access the endpoints of the Supervisor Web Server (SWS) based on their role.
2. IoT platforms are authenticated in the consortium ledger by applying a simplified version of Model 2 of the IAA component (i.e. a smart contract that handles authorisation requests), since no payments are considered in the pilot case. Once an IoT platform is registered in the consortium ledger (i.e. its wallet address is recorded), the smart contract acts as an authorisation contract to confirm it has the authority to perform certain transactions.

Privacy and Data Sovereignty

The privacy and data sovereignty component provides tools for addressing data privacy in the implemented use cases and in the actors' activities. In particular, the following policies have been implemented:

- The SWS is a full node of the consortium ledger, which is responsible for providing wallet functionality to the federated IoT platforms.
- Every transaction fired by an IoT platform to the consortium ledger is digitally signed using the platforms' private key.
- Upon registration of an actor into the FSC web application, a unique ID and role are assigned to it. Only the role information is included in the transactions written in the consortium ledger, and the mapping between actors' profiles, IDs and roles is accessible only by the SOFIE system administrator.
- In the preparation of data that relates to multiple actors (possibly belonging to different segments) which should be combined together, actor IDs are used to retrieve and chain information from the consortium ledger.

Semantic Representation

In the FSC pilot, three IoT platforms are federated into the system architecture. For each platform, an adaptation layer is implemented to interface the Supervisor Web Server (SWS) and to expose the platform's functionality and the things' services according to the same semantics, thus enabling cross platform access and interoperability. The SR component also allows for different IoT platforms to describe their own features and capabilities.

Federation Adapters

Each federated IoT platform is connected to the pilot platform through a federation adapter as described in Section 9.1. FAs allow for a non-intrusive way to leverage on existing (legacy) IoT platforms.

Key benefits

The key benefits for the FSC pilot from using the Framework components include:

- The development of a decentralised and immutable data management and storage framework that leverages multiple DLTs using IL component to enable automated, reliable, and flexible operation of all critical data used in the food supply chain. It provides a similar level of trust for the stored data compared to storing all data to a public ledger while radically reducing the associated costs as most operations take place on a private ledger. The use of DLTS also facilitates the development of further enabling services, e.g. by using add-on modules, processes or smart contracts to enable automated payments, trust evaluation, etc. among the parties.
- The transparent federation of heterogeneous IoT silos used by the involved parties in a technology-agnostic way that enables cross-platform interoperability. As a result, applications or services can discover resources from different IoT platforms through the same interfaces and by using the same formats to communicate data.
- The provision of secure data access and retrieval services in the sense of data confidentiality and integrity for all involved parties based on their role in the supply chain (e.g. viewing where the produce comes from, which steps it passes through, which other produce may be affected in the case of quality issues etc.).
- The ability to replace any partner that has been part of the supply chain but for any reason has paused his operation with another partner that is already registered in the pilot platform that has a proven record of offering high quality services.

10.2 Decentralised Energy Flexibility Marketplace Pilot

The Decentralised Energy Flexibility Marketplace (DEFM) pilot is centered around the SOFIE Marketplace component, which is used to create a decentralised marketplace, in which a Distribution System Operator (DSO) and Fleet Managers may request and offer energy flexibility in order to balance the excess of production on a local grid. The marketplace is complemented by the IL and SR components: the first one supports the marketplace by increasing the trustworthiness of the internal private DLT by recording its key events to an external public DLT, while the latter is used to analyse on-the-fly the data gathered from the smart meters, ensuring their consistency with the expected model before making a new market request based on them.

The pilot also demonstrates the usage of IoT smart meters to obtain accurate forecasts and live data from the pilot site. The gathering, persistence, and analysis of the data produced from the smart meters is facilitated by the DEFM Federation Adapter (FA).

Figure 10.2 presents the architecture and the SOFIE framework components used in the pilot. The framework components are leveraged in the following ways:

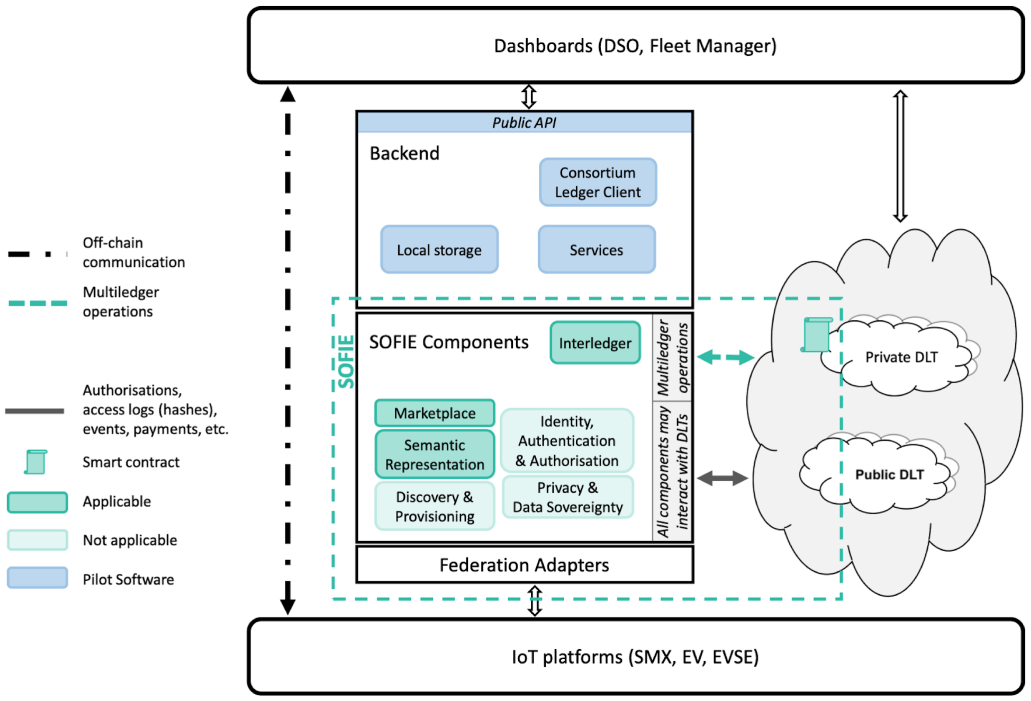

Figure 10.2: DEFM pilot architecture.

Marketplace

The Marketplace component is used to run an Ethereum based decentralised marketplace. The requirements for the marketplace for the energy flexibility use case were defined already in the early phases of the project, and the latest implementation of the component is now deployed on the pilot site. The *request* side on the marketplace is managed by the DSO operator, while the Fleet Managers control the *offer* side. Each Demand Response campaign starts with a new flexibility request on the market created by the DSO. Each request is characterised by its deadline, delivery window (start and end times), requested amount, network zone and, finally, the maximum number of *tokens* that the DSO will pay to the winning

offer. The Fleet Managers receive the open requests on the marketplace and can decide to participate in an open campaign offering their availability to charge its electric vehicles using the charging stations in the specified zone during the specified time window. Each offer also includes the number of tokens required by the Fleet Manager as the reward. The smart contract defining the rules of the marketplace also includes the logic used to select the winning offer: a *reverse* auction is used, so the lowest bidding fleet manager wins.

The tokens are not transferred until the delivery window starts: at this point, the smart contract can be queried to finalise the payment. The delivery is validated by using the charging station APIs and checking the recharged energy against the quantity agreed on the market request. When the required amount has been recharged, the payment is finalised and the request is closed successfully.

Thanks to the Marketplace component, the DSO and the Fleet Manager can operate over a decentralised marketplace without the need to define a smart contract from scratch: the marketplace, together with its backend, includes the generic interfaces to be implemented to develop a marketplace smart contract. The interfaces define the basic functionalities (e.g. the requests/offers matchmaking process or the authorisation policies) and attributes (e.g. status indicators, quantities associated with the requests, number of tokens for each request/offer) to ensure that each phase can be completed as intended without inconsistencies.

Semantic Representation
The DEFM pilot uses live data from the smart meters on the field together with forecasts calculated using historical data to determine the need by the DSO for creating a new flexibility request on the marketplace and the quantity needed for each request. For this reason, it is important to verify the integrity of the data used on the application. The semantic representation component is used to validate on-the-fly the data retrieved by the application against the data model before displaying it on the web application.

Interledger
The DEFM pilot uses a private Ethereum ledger to run the smart contracts powering the decentralised marketplace and store the related data. The Interledger is then used to store the most important events for each request to a public ledger. The life cycle of a market request is characterised by different stages as reported above in the marketplace section: the selection of the winning offer and the delivery phase are the two mandatory steps which separate a request closed and paid successfully from an incomplete request. For this reason the interledger component is used to broadcast the two events to a public ledger to maintain an immutable copy of the record, for validation and auditability purposes.

Federation Adapter
The DEFM FA was presented in detail in Section 9.2. It enables the management of the IoT devices and the whole lifecycle of context data. In this way the platform is able to consume context data from the smart devices in the pilot while remaining decoupled from the data sources. This allows the addition of new data sources and the extension of the platform to include new consumer applications without the need to reconfigure the platform.

Key Benefits
The key benefits for the DEFM pilot from using the Framework components include:

- Easy development of a decentralised marketplace using smart contracts on Ethereum ledger to manage the platform. This enables auditability and automated payments among the parties in a trustworthy and tamper proof environment.

- The ability to run over a combination of private and public ledgers thanks to the IL component. This, as the pilot emulation has evaluated, allows the marketplace to save significantly in execution costs while maintaining a high level of trustworthiness.
- The technology-agnostic federation of different IoT platforms.
- The ability to validate the data feed from the smart meters before reaching the end user's application, ensuring that the marketplace requests are based on valid data.
- The possibility for new partners to join the marketplace at any time and start providing flexibility services, thus enhancing competition and the quality of the services offered.

10.3 Context-Aware Mobile Gaming Pilot

The focus of the mobile gaming pilot shown in Figure 6 is to explore how DLTs can be used to provide new gaming features for players, as well as to validate the potential of location-based IoT use cases. The gaming pilot also leverages four SOFIE components to provide new gaming features and enhance player experience in the following ways:

Interledger

The Mobile Gaming pilot uses the SOFIE Interledger component to communicate between the permissioned Hyperledger Fabric and public Ethereum networks. Multiple games share and store in-game assets on a permissioned Hyperledger Fabric ledger and use interledger to transfer those assets to public Ethereum, where they can later be sold on the marketplace. The component helps link the closed ecosystem of games and game developers to the public ecosystem of trading games and other virtual assets. Transferring of the assets is achieved by calling smart contracts that are emitting and receiving events.

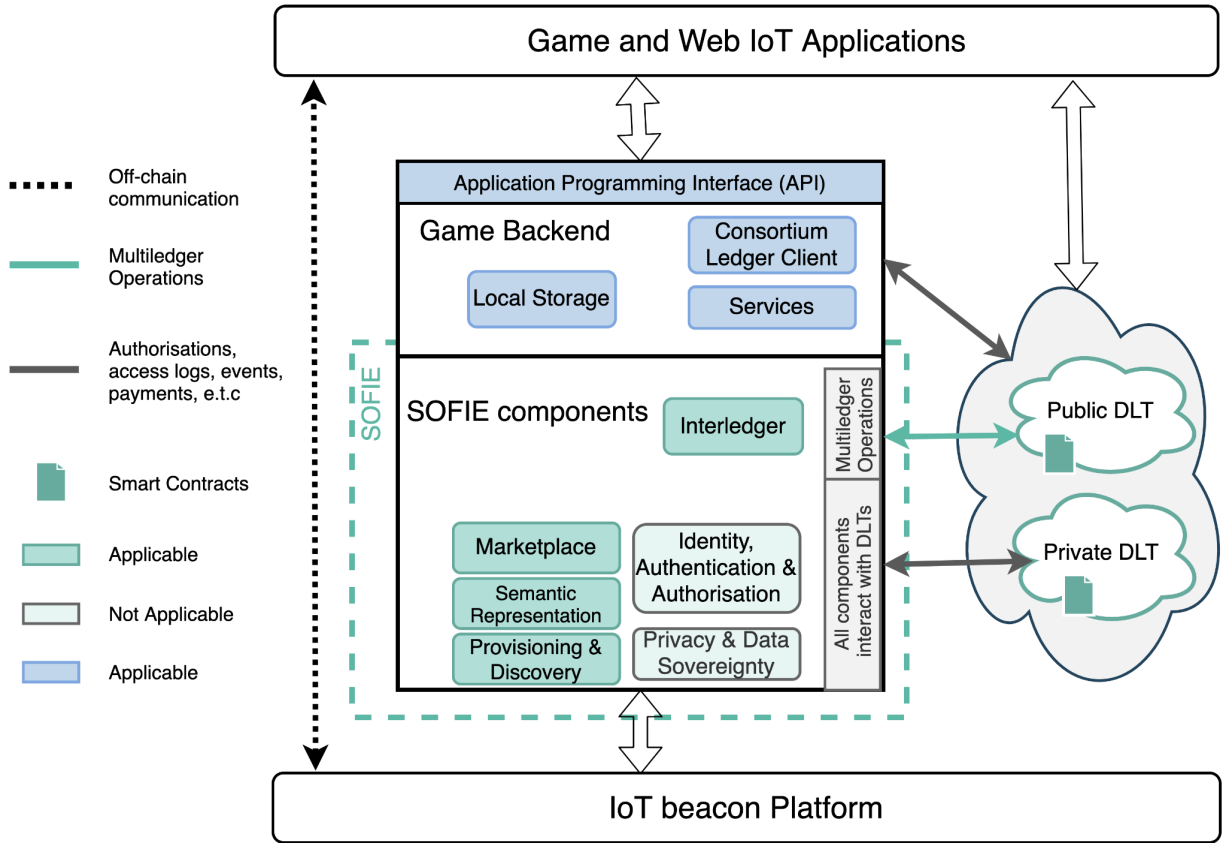

Figure 10.3. Mobile Gaming Architecture with SOFIE components

Provisioning and Discovery

The mobile gaming pilot uses the Provisioning and Discovery component for gamifying the process of discovering, configuring and maintaining large IoT beacon deployments in the scavenger hunt game. The component scans nearby IoT devices, learns their capabilities, and determines which devices are suitable for provisioning based on the requirements set by game developers. After provisioning each IoT device, the component configures it to work as a BLE beacon that can be used as a Point-of-Interest (PoI) in the scavenger hunt prototype.

Semantic Representation

The mobile gaming pilot uses the Semantic Representation component to describe the devices using WoT-Things Descriptions. The semantic file is encoded in a JSON format that also allows JSON-LD processing. Using this component, the IoT devices can describe their capabilities that can be later used to provision the device.

Marketplace

The mobile gaming pilot uses the Marketplace component for the trading of in-game assets. Players can place bids for virtual assets on the public Ethereum ledger. The component enables the actual trade of resources in an automated, trusted, and decentralised way. Once digital assets have been stored on the ledger, the ownership and the item itself cannot be altered. DLTs also help maintain the scarcity of a virtual item in a secure and verified way.

Key Benefits

The key benefits for the CAMG pilot from using the Framework components include:

- The Interledger component helps to achieve interoperability between different distributed ledgers and could be used to create new business opportunities as closed platforms can be connected securely and transparently. Using Interledger also helps to get benefits of specific ledgers: Fabric for throughput and privacy, and Ethereum for security and traceability.
- The Provisioning and Discovery component takes advantage of the already deployed Web of Things (WoT) devices. It also creates new business opportunities for device holders, namely micropayments for services used, and the whole process of discovering and provisioning can be trusted and automated through DLTs.
- The Semantic Representation component helps unify different WoT devices used as beacons and also provides service interoperability.
- Using the marketplace component, the in-game assets can have some real-world value also, e.g. trade a skin with electricity. The DLT-based marketplace grants security, transparency, and traceability, with the effect of increasing a healthy competition among the players participating.

The gaming pilot leverages SOFIE components to provide new gaming features and enhance player experience as described in. Table 10.2

Table 10.2. SOFIE components used and their benefits for Context Aware Mobile Gaming Pilot

SOFIE Component	Usage	Benefit
Interledger	• Communication between the permissioned Hyperledger Fabric and public Ethereum networks. • Linking the closed ecosystem of games and game developers to the public ecosystem of trading games and other virtual assets. • Achieved by calling special smart contracts and emitting and receiving events	• Interoperability between different distributed ledgers. • New business opportunities as closed platforms can be connected. • Benefits of specific ledgers: Fabric for throughput and privacy, and Ethereum for security and traceability.
Provisioning and Discovery	• Scans nearby IoT devices, learns their capabilities, and determines which devices are suitable for provisioning. • The component configures the IoT devices to work as BLE beacons for the Scavenger Hunt game prototype. • Provisioned devices can be used in the creation of new Hunts in the game. • Permissions to configure devices can be managed through DLTs	• Take advantage of the already deployed Web of Things (WoT). • New business opportunities for device holders, namely micropayments for services used. • Can be trusted and automated through DLT.
Semantic Representation	• Semantic description of devices using WoT-Things Description. • Used along with the Discovery and Provisioning component. • Encoded in a JSON format that also allows JSON-LD processing	• Devices can describe their own capabilities. • Service interoperability. • Unify different WoT devices used as beacons.
Marketplace	• Players can place bids for trading in-game virtual assets. Enables the actual trade of resources in an automated, trusted, and decentralised way. • Once digital assets have been stored on the ledger, the ownership and the item itself cannot be altered. • DLTs also help maintain the scarcity of a virtual item in a secure and verified way.	• In-app assets have real world value, e.g trade skin with electricity. • DLT-based marketplace grants security, transparency, and traceability, with the effect of increasing a healthy competition among the players participating.

10.4 Decentralised Energy Data Exchange Pilot

As described in Section 9.3, the Decentralised Energy Data Exchange (DEDE) pilot has integrated all the Framework components it uses into the thick DEDE FA. The following components have been utilised:

- *Privacy and Data Sovereignty* - It is up to the service providers to say, which attributes they want their clients to prove about themselves. By default, the protocol expects the attributes to be proven with verifiable credentials, but the PDS component offers a

simpler alternative by issuing a JSON Web Token (JWT) that proves certain attributes about the holder of the token. This token can be sent together with a request and thus avoid the extra round trip for proof request.

- *Identity, Authentication, Authorisation* - This is the counterpart component for the PDS component, if JWT based proved attributes are used. If the client sends a JWT together with a service request, the IAA component can verify this token. In this case, the authorisation decision can be made without an extra round trip to the client requesting proof of credentials.
- *Interledger* - Each node can increase their trust for the Hyperledger Indy instance by periodically recording its state in the KSI blockchain. The interledger component takes care of this. It is not strictly required for the protocol to work, but can serve as an additional tamper-proofing mechanism for private Hyperledger Indy deployments.

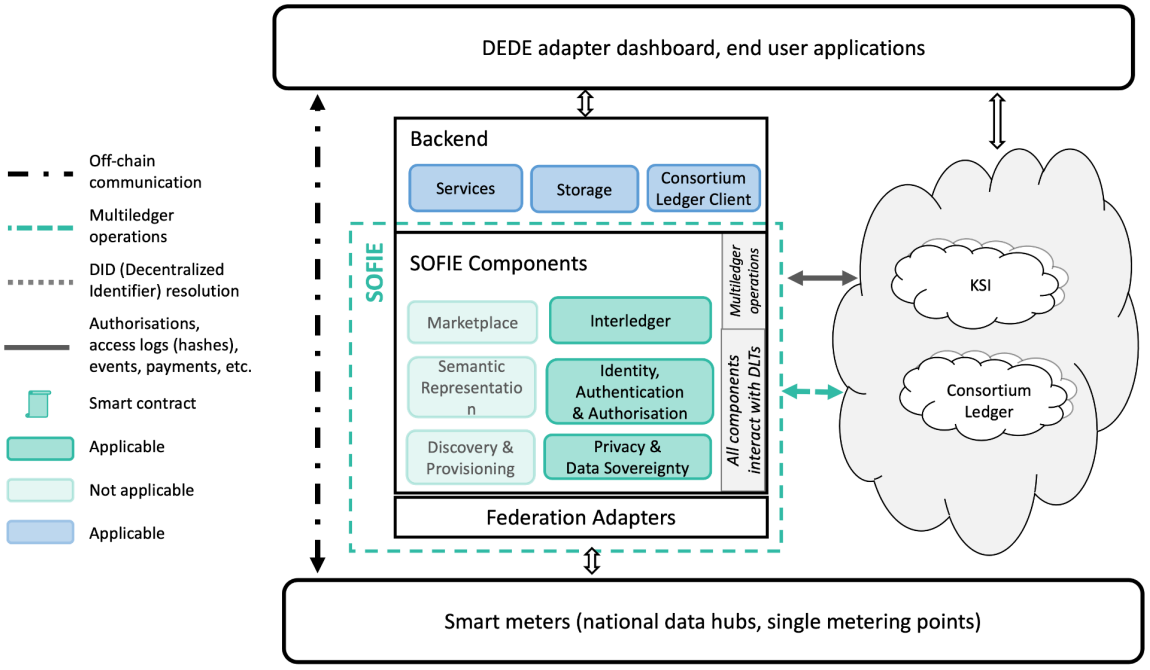

Figure 10.4. Decentralised Energy Data Exchange Pilot with SOFIE components

Key Benefits

The key benefits for the DEDE pilot from using the Framework components include:

- The Privacy and Data Sovereignty component supports different credential and token types making it easier to integrate different systems with the DEDE FA.
- Using tokens with the IAA component speeds up the access control process as extra round trips can be avoided.
- Interledger enables higher trustworthiness levels by anchoring information on private ledgers to public ledgers.

10.5 SMAUG

SMAUG - Secure Marketplace for Access to Ubiquitous Goods - is a decentralised, blockchain-based auction marketplace for the rental of smart lockers. The marketplace mediates the interactions between Smart Locker Owners (SLO) and potential Smart Locker Renters (SLR), and at the same time provides decentralisation (anyone can participate in the marketplace interactions), auditability (all the steps of the rental process are immutably logged

on the blockchain), and security (the smart contract logic powering the marketplace is secured by the blockchain consensus protocol).

SMAUG was developed as a reference application demonstrating how all SOFIE components can easily be leveraged to create a fully functional application. The usage of the different components is described below.

Marketplace

The Marketplace component forms the core of SMAUG: it enables the interactions between SLOs and SLRs, and ensures the safety of the marketplace ecosystem. A representation of the different entities participating in the marketplace interactions is given in Fig. 10.4. The smart contract allows SLOs to create auctions, which optionally can also accept instant-rent offers, i.e. offers that aim to rent a locker for the specified slot of time thus bypassing the auction process, though typically for a higher rent. SLRs can present offers for those auctions by escrowing enough Ethers to cover the whole rental time requested. If access is granted to the requester, the escrowed Ethers will be claimed by the SLO, thus increasing the SLO's total balance. If, on the other hand, the offer submitted is not among the auction winners, the offer creator will be notified to reclaim back the money previously escrowed, thus leaving the balance unchanged.

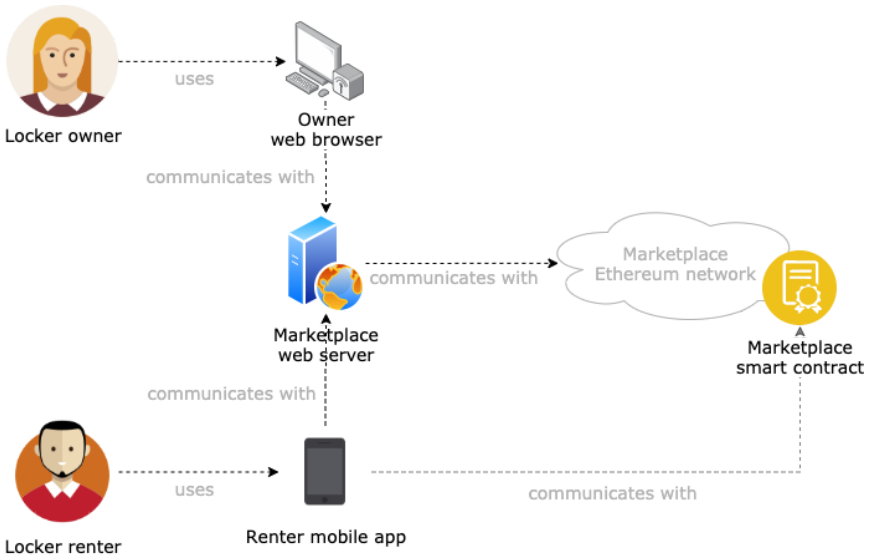

Figure 10.4: The entities participating in the marketplace interactions.

Interledger

The SOFIE Interledger (IL) component is used for *bi-directional* data transfer between the two ledgers on which SMAUG relies. The **marketplace-to-authorisation** data transfer happens when an auction is closed, and the information about the winning offers is transferred to the authorisation ledger, where further processing is performed to generate the access tokens the winning SLRs will use to access the smart locker. The **authorisation-to-marketplace** data transfer happens, when the access tokens for all the winning offers of a given auction have been generated, and the encrypted access tokens are transferred back to the marketplace blockchain, which will then notify the interested actors about the new token. The different steps performed during the two Interledger flows are shown in Fig. 10.5.

Identity, Authentication, and Authorisation

The Identity, Authentication, and Authorisation (IAA) component is used in different parts of SMAUG. It is used by the SMAUG marketplace backend, as depicted in Fig. 10.4, to validate

JWTs used by SLOs when interacting with the marketplace management interface. Furthermore, IAA is also used by the smart locker (SL) to verify the validity of the access token presented by the winners of the auctions when they want to access the smart locker. This token is generated by the smart locker access manager, as shown in Fig. 10.5, and then decrypted by the SLRs after they have been notified by the marketplace blockchain.

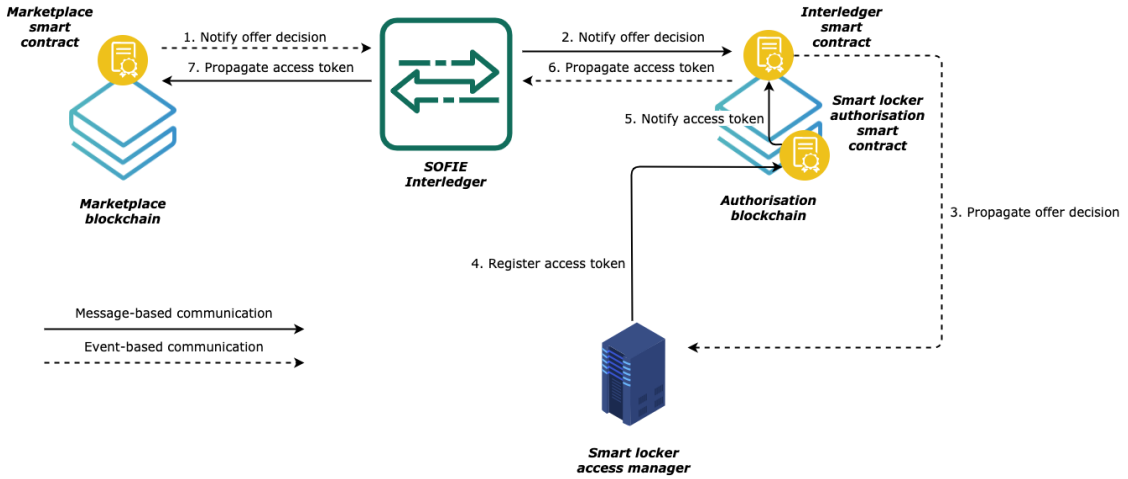

Figure 10.5: The different steps performed during an Interledger process.

Privacy and Data Sovereignty

The Privacy and Data Sovereignty component is used in two parts of SMAUG. It is used by the SMAUG marketplace backend to support DID-based authentication of SLOs accessing the web management interface. Specifically, PDS allows users to authenticate themselves using a DID previously registered on the platform, and upon successful authentication, PDS generates a JWT that the SLOs will successively use to perform the actions the management interface provides. The JWT issued by PDS is then verified by IAA upon presentation by the SLOs, as explained in the section about IAA.

The other part of SMAUG that uses PDS involves the smart locker access manager and the authorisation smart contract that is used to log and store the access tokens the smart locker access manager generates. Upon receiving Interledger information originating from the marketplace blockchain, the smart locker access manager utilises the PDS component to generate the access token and to log it on the authorisation blockchain. Once the process is completed, the smart locker access manager triggers an Interledger event to communicate the value of the access token(s) generated back to the marketplace blockchain.

Semantic Representation

The SOFIE Semantic Representation (SR) component is used in SMAUG to validate locker information that SLOs upload when registering a new locker. In order for SMAUG to be an open and decentralised smart locker marketplace, all smart lockers must follow a common representation. This is made possible by SR, and the SMAUG marketplace backend delegates to SR the validation of the new lockers that SLOs want to add to the SMAUG ecosystem.

Provisioning and Discovery

The SOFIE Provisioning and Discovery (PD) component allows smart lockers to be discoverable by interested nearby customers using the Bluetooth Low Energy (BLE) protocol. PD and SR together give SMAUG the unique properties of decentralisation and openness, allowing anyone willing to be part of the SMAUG ecosystem to 1. register the locker on the

marketplace by using the SMAUG semantic representation, and 2. to allow the locker to be discoverable by integrating the SOFIE PD component.

Key Benefits

The key benefits for SMAUG from using the Framework components include:

- With the Marketplace component, all the logic is embedded in the smart contract, which guarantees that the business rules are always enforced by the marketplace.
- The usage of Interledger is fundamental as it allows the system to make use of multiple blockchains, where each blockchain might offer different properties (as in the case of permissionless vs. permissioned DLTs), while still ensuring the safety and atomicity of the bridging process.
- The main benefit of using PDS in SMAUG is the possibility to support DID-based authentication out of the box, and to uncouple the marketplace and authorisation blockchains by delegating access token generation and logging to several, potentially independent entities - the smart locker access managers.
- The IAA component then provides support for simple implementation of the access control based on the tokens.
- The Semantic Representation enables dynamic registration of new lockers by ensuring they follow the required representation.
- The Provisioning and Discovery component enables automatic discovery of nearby smart locker, thus functioning as a distributed directory of lockers.

11 Summary

The SOFIE Framework provides example implementations of all the SOFIE components and federation adapters. The components are all utilised in the SOFIE pilots and in the reference application SMAUG, thus ensuring their maturity: the P&D component reached TRL 6, while *all other components and FAs reached TRL 7*.

The *Interledger (IL)* component is the backbone of the SOFIE Framework, enabling easy federation over different ledgers. The component supports multiple ledger types and support for new ledgers can be easily added. The provided atomic transaction functionality can be leveraged to different types of applications as demonstrated by the multiple examples provided.

The *Identity, Authentication and Authorisation* (IAA) component can be used to secure any HTTP-based resource with minimal effort. IAA operates transparently and requires no modification to the protected resource. Similarly, the only modification required to client applications is the addition of an HTTP header. IAA can be used with a variety of authentication and authorisation methods, including emerging standards such as Decentralised Identifiers (DIDs) and Verifiable Credentials (VCs), as well as with traditional mechanisms such as bearer tokens. Furthermore, it can be easily integrated to an OAuth2.0 workflow. Finally, IAA can be configured with rich access control policies using JSON path.

The *Privacy and Data Sovereignty* component extends OAuth2.0 to cover additional authorisation grants, including VCs and DIDs. Furthermore, the PDS authorisation server supports blockchain-based tokens, as well as logging to smart contracts. The privacy module of PDS allows for local differential privacy, which protects the end-users even from curious system operators. Additionally, a PDS smart contract enables fair exchange of privacy preserving responses and service fees: a data consumer pays service fees only if the appropriate number of responses has been collected, but cannot access the provided responses before it pays.

The *Semantic Representation* component can be used to easily open the system data model to external parties. The implementation of this component allows the system to be used by other systems or IoT devices automatically, without the need of manual system integration. The component automatically validates the messages external systems are sending and can lower the effort on implementing security rules by integrating them in the data model. This functionality allows the system using the SR component to be reachable to everyone satisfying the security restrictions.

The *Marketplace* component provides an open decentralised marketplace that anyone can join. It supports different pricing models for auction and fixed priced sales, and covers the whole sales process from creating the auction all the way to the buyer reporting successful receipt of the item thus supporting comprehensive transparency and accountability for the trading.

The *Provisioning and Discovery* component can be used to discover new IoT devices using multiple discovery protocols with minimal effort. The component provisions the discovered devices based on the user requirements and it also configures the IoT devices. The component uses the SR component for providing the metadata to the IoT device. It uses Bluetooth and DNS as discovery protocols and can also be used with the already deployed beacons.

12 References

[D2.5] Y. Kortesniemi et al. “SOFIE Deliverable 2.5 - Federation Framework, 2nd version”, December 2019. Available at: https://www.sofie-iot.eu/results/project-deliverables

[D2.6] Y. Kortesniemi et al. “SOFIE Deliverable 2.6 - Federation Architecture, final version”, October 2020. Available at: https://www.sofie-iot.eu/results/project-deliverables

[D5.3] I. Oikonomidis et al. “SOFIE Deliverable 5.3 - End-to-end Platform Validation”, July 2020. Available at: https://www.sofie-iot.eu/results/project-deliverables

[D5.4] I. Oikonomidis et al. “SOFIE Deliverable 5.4 - Final Validation & Replication Guidelines”, December 2020. Available at: https://www.sofie-iot.eu/results/project-deliverables

[FAs] SOFIE Federation Adapters, open-source implementations of the FAs used in the SOFIE pilots, available at: https://github.com/SOFIE-project/Federation-Adapters

[Fot2020] N. Fotiou, I. Pittaras, V.A. Siris, S. Voulgaris, G.C. Polyzos, "OAuth 2.0 authorization using blockchain-based tokens," Proceedings of the NDSS 2020 Workshop on Decentralized IoT Systems and Security (DISS), San Diego, CA, USA, 2020

[Framework] SOFIE Framework, an open-source software implementation of the SOFIE Architecture, available at: https://github.com/SOFIE-project/Framework

[Sir2019] V.A. Siris, P. Nikander, S. Voulgaris, N. Fotiou, D. Lagutin, G. Polyzos: Interledger Approaches. In: *IEEE Access* Bd. 7 (2019), S. 89948–89966