Innovation Action
H2020-SU-DS-2018-2019-2020 (Digital Security)
Topic: SU-DS04-2018-2020

PHOENIX

Electrical Power System's Shield against complex incidents and extensive cyber and privacy attacks

Deliverable D2.3

Secure and Persistent Communications Layer (Ver. 1)

Authors	Nikolaus Wirtz (RWTH), Alessio Bianchini, Elena Sartini, Luigi Briguglio, Carmela Occhipinti (CEL), Timotej Gale, Blaž Merela, Tomaž Bračič, Denis Sodin (CS), Dimitrios Skias (INTRA), Maryam Pahlevan (AALTO), Wafa Ben Jaballah, Cyrille Piatte (TSG), Artemis Voulkidis (SYN), Tomaž Dostal (ISKRA)
Nature	Report
Dissemination	Public
Version	1.1
Status	Final
Delivery Date (DoA)	30.04.2021
Actual Delivery Date	30.04.2021

Keywords	Secure and Persistent Communication, Cyber Threat Information, Distributed Ledger Technologies, federated cloud, resilience, Electrical Power and Energy System
Abstract	The PHOENIX project focuses on the protection of European Electrical Power and Energy System (EPES) assets and networks against cyber-attacks. The Secure and Persistent Communications (SPC) Layer is one of the key components to achieve this goal, enabling coordinated cybersecurity measures and the secure exchange of Cyber Threat Intelligence at the same time enforcing privacy. To increase the availability and reliability of the

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 832989

	PHOENIX core components and foster communications resilience, the platform deployment of PHOENIX follows the cloud native paradigm with the adoption of container-based operation and container orchestration processes. The SPC Layer offers security over legacy protocols currently used in EPES infrastructures as well as data persistency by adopting a data-centric approach based on federated Distributed Ledger Technologies to achieve a higher degree of persistency, traceability, availability, integrity, and interoperability in the context of data communications. Furthermore, several “by-design” options to increase the resilience of EPES systems have been investigated. This deliverable provides the second version of the SPC Layer description whereas the final update will be given in D2.4: Secure and Persistent Communications Layer (Ver. 2).

DISCLAIMER

	Participant organisation name	**Short**	**Country**
01	Capgemini Technology Services	CTS	France
02	THALES SIX GTS FRANCE SAS	TSG	France
03	THALES Research & Technology S.A.	TRT	France
04	SingularLogic S.A.	SiLO	Greece
05	DNV-GL AS	DNV	Norway
06	INTRASOFT International S.A.	INTRA	Luxemburg
07	Iskraemeco	ISKRA	Slovenia
08	Atos SPAIN SA [Terminated]	ATOS	Spain
09	ASM Terni	ASM	Italy
10	Studio Tecnico BFP srl	BFP	Italy
11	Emotion s.r.l.	EMOT	Italy
12	Elektro-Ljubljana	ELLJ	Slovenia
13	BTC	BTC	Slovenia
14	Public Power Corporation S.A.	PPC	Greece
15	E.ON Solutions Gmbh [Terminated]	EON	Germany
16	Delgaz Grid SA	DEGR	Romania
17	Transelectrica S.A.	TRANS	Romania
18	Teletrans S.A.	TELE	Romania
19	Centro Romania Energy	CRE	Romania
20	CyberEthics Lab	CEL	Italy
21	GridHound GmbH [Terminated]	GRD	Germany

22	Synelixis Solutions S.A.	SYN	Greece
23	ComSensus	CS	Slovenia
24	AALTO-KORKEAKOULUSAATIO	AALTO	Finland
25	Rheinisch-Westfälische Technische Hochschule Aachen	RWTH	Germany
26	Capgemini Consulting [Terminated]	CAP	France
27	ATOS IT Solutions and Services Iberia SL	ATOS IT	Spain
28	DNV GL NETHERLANDS B.V.	DNV-NL	Netherlands

ACKNOWLEDGEMENT

This document is a deliverable of PHOENIX project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nº 832989.

The opinions expressed in this document reflect only the author's view and in no way reflect the European Commission's opinions. The European Commission is not responsible for any use that may be made of the information it contains.

Document History

Version	Date	Contributor(s)	Description
V0.1	08.02.2021	RWTH	Initial ToC
		TSG, AALTO, CS, INTRA, RWTH	Refinements of the ToC
V0.2	15.02.2021	RWTH	Consolidation of the ToC
		TSG, AALTO, CS, INTRA, RWTH	1st round of contributions
V0.5	15.03.2021	RWTH	Consolidation of the 1st round of contributions
		INTRA, AALTO, CS, CEL, TSG, RWTH	2nd round of contributions
V0.7	16.04.2021	RWTH	Consolidation of the 2nd round of contributions
		TSG, SYN, ISKRA	Additional inputs
V0.9	22.04.2021	RWTH	Consolidation
	27.04.2021	SYN, CS, RWTH	Internal review and peer review
V1.0	30.04.2021	RWTH, TSG, AALTO, INTRA, ISKRA, CEL	Consolidation of review feedback and finalization
V1.1	30.06.2021	RWTH	Updates on the basis of recommendations following the midterm review – fixed references

Document Reviewers

Date	Reviewer's name	Affiliation
26.04.2021	Artemis Voulkidis	SYN
27.04.2021	Timotej Gale	CS
27.04.2021	Hendrik Flamme	RWTH

Table of Contents

Definitions, Acronyms and Abbreviations 9

Executive Summary 11

1. Introduction 13

2. Semantics and Ontologies – Application to the PHOENIX environment 14

2.1. TAXII Server 14

2.1.1. PHOENIX Pub/Sub Enhancements in TAXII Server 14

2.1.2. TAXII Server in the SPC Layer 16

2.1.3. Authentication on TAXII Server 19

2.1.4. Ledger-specific adapter for TAXII Server 19

2.2. TAXII Client 19

3. Transaction and Communication 20

3.1. Cloud Native for 5G communications 20

3.1.1. Cloud Native Paradigm 20

3.1.2. Service Mesh 21

3.1.3. Service Mesh Components & Capabilities 21

3.2. Deployment based on the (federated) cloud 23

4. Federated ledger & Interledger on SPC Layer 25

4.1. Distributed Ledger Technologies and Interledger solutions 25

4.2. Interledger solution 26

4.2.1. Interfaces and Services 27

4.2.2. Internal Architecture 29

5. Initial communication platform 35

5.1. Collecting data from USG 36

5.2. Interactions with Interledger solution 36

5.3. Smart Contract Management and Interactions with PPE 38

6. Resilience Enhancement Methodology 41

6.1. Use case 41

6.1.1. Relation to ASM threat scenarios 41

6.2. Design and implementation of the Resilience Enhancement Methodology 42

6.2.1. Functions 42

6.2.1. Double Virtualization 44

6.2.2. Implementation of the Resilience Enhancement Methodology 47

6.3. Outlook 48

7. Simulation Studies 49

7.1. Laboratory set-up 49

7.1.1. Grid model 49

7.1.2. REM infrastructure 49

7.2. Evaluation 50

7.2.1. Scenario 51

7.2.2. Criteria 51

8. Conclusion 52

9. References 53

List of figures

Figure 1: TAXII Channels for CTI exchange in a publish/subscribe manner [5] 14
Figure 2: The publish/subscribe exchange mechanism in PHOENIX. 15
Figure 3: Sequence diagram for the write operation of TAXII data on TAXII server and SPC ledger.... 16
Figure 4: Sequence diagram for reading TAXII data from the TAXII server and SPC ledger. 18
Figure 5: Generic Kubernetes concepts architecture used in PHOENIX. 23
Figure 6: Structure of the SPC DLT components (deployments and pods) under the SPC namespace of Kubernetes. 24
Figure 7: Structure of the Kubernetes pods regarding the TAXII-related components. 24
Figure 8: Kubernetes pods governing the cloud-storage configuration of PHOENIX. 24
Figure 9: Relationship between PHOENIX components and DLTs. 27
Figure 10: The Interledger component workflow [46].. 27
Figure 11: Interledger component architecture [46]. 29
Figure 12: One-to-one mode of the Interledger component [46]. 30
Figure 13: Multi-ledger mode of Interledger component [46]. 31
Figure 14: The class diagram of the IL component [46].. 32
Figure 15: Architecture of Decentralised Interledger [46]. 33
Figure 16: Example of Interactions of the SPC Layer with the TAXII clients. 35
Figure 17: USG positioning in the SPC Layer. 36
Figure 18: An example scenario of usage of the IL component within SPC Layer. 37
Figure 19: PPE Deployment Diagram. 38
Figure 20: PPE Class Diagram. 40
Figure 21: DV use case example - grid monitoring based on PMU measurements. 44
Figure 22: DV use case example - PDC failure without DV. 45
Figure 23: DV use case example - PDC failure with DV. 45
Figure 24: Simplified OODA Diagram 46
Figure 25: Simplified REM infrastructure and laboratory set-up. 50

List of tables

Table 1: Interledger interfaces [46]... 28
Table 2: DV Asset - overview of relevant flows 48
Table 3: Specification of single board computers for REM deployment. 49
Table 4: Criteria for evaluation on the REM implementation... 51

Definitions, Acronyms and Abbreviations

AAM	Accountability & Access Management
API	Application Programming Interface
CERT	Computer Emergency Response Team
CNA	Cloud Native Application
CTI	Cyber Threat Intelligence
DIL	Decentralised Interledger
DLT	Distributed Ledger Technologies
DN	Distribution Network
DSO	Distribution System Operator
DV	Double Virtualization
EPES	Electrical Power and Energy System
FLA	Fault detection and Localization Algorithm
FLISR	Fault Localization, Isolation and Service Restoration
GDPR	General Data Protection Regulation
HILP	High Impact Low Probability
HTLC	Hashed Time Locked Contract
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I2SP	Incident Information Sharing Platform
IDE	Integrated Development Environment
IED	Intelligent Electronic Device
IL	Interledger
ILP	Interledger Protocol
IMEC	Incident Mitigation & Enforcement Countermeasures
LSP	Distributed Ledger Technology
MV	Medium Voltage
OODA	Observe-Orient-Decide-Act
PDC	Phasor Data Concentrator
PMU	Phasor Measurement Unit
PoA	Proof-of-Authority
PoW	Proof-of-Work
PQM	Power Quality Meter
PPE	Privacy Protection Enforcement
REM	Resilience Enhancement Methodology

RES	Renewable Energy Source
SAIDI	System Average Interruption Duration Index
SAIFI	System Average Interruption Frequency Index
SAPC	Situation Awareness, Perception & Comprehension
SC	Smart Contract
SCADA	Supervisory Control and Data Acquisition
SE	State Estimation
SM	Smart Meter
SPC	Secure and Persistent Communications
SR	Service Restoration
SSO	Single Sign-On
STIX	Structured Threat Information eXpression
TAXII	Trusted Automated Exchange of Intelligence Information
TLS	Transport Layer Security
USG	Universal Secure Gateway

Executive Summary

PHOENIX focuses on the protection of European Electrical Power and Energy System (EPES) assets and networks against cyber-attacks. The Secure and Persistent Communications (SPC) Layer is one of the key components to achieve this goal, enabling coordinated cybersecurity measures and the secure exchange of Cyber Threat Intelligence (CTI), at the same time ensuring privacy. An initial description of the SPC Layer was given in deliverable D2.1: PHOENIX platform architecture specification [1] as part of the PHOENIX architecture description and was refined and expanded by the requirements of the SPC Layer in deliverable D2.2: Secure and Persistent Communications Layer (Ver. 0) [2]. This deliverable provides an update to D2.2.

The application of semantics and ontologies to the PHOENIX environment is a key concern for the SPC Layer. The approach adopted in the project is built on the utilization of standard protocols for cyber threat intelligence protocols and services. Furthermore, ledger-specific adapters ensure compatibility with the federated ledger and Interledger (IL) approach of PHOENIX.

To increase the availability and reliability of the PHOENIX core components and foster communications resilience, the platform deployment of PHOENIX follows the cloud native paradigm with the adoption of mostly stateless, container-based operation and relevant container orchestration processes. Consequently, following the cloud native paradigm, Cloud Native Applications and Service Mesh for 5G communication have been explored for the PHOENIX environment.

In the PHOENIX platform, the SPC Layer offers security over the legacy protocols that are currently used in EPES infrastructures as well as data persistency. The SPC Layer adopts a data-centric approach based on federated Distributed Ledger Technologies (DLT) to achieve a higher degree of persistency, traceability, availability, integrity, and interoperability in the context of data communications. Beyond usage of DLTs within the SPC Layer, different components of the PHOENIX platform can also leverage the use of DLT as a persistent and immutable data storage. Towards enabling the utilization of multiple ledgers simultaneously within the PHOENIX platform, a distributed and resilient solution for interconnecting varying ledger networks has been designed and implemented, which is also suitable for critical Electrical Power and Energy System assets and data. The current implementation of the Interledger component enables connections between various Distributed Ledger Technologies and can be easily extended to support other ledger networks. This document specifies the interfaces and services as well as internal architecture.

Leveraging on the above activities, the SPC Layer provides secure federated communications among PHOENIX components at the level of Large-Scale Pilots as well as between those components and the Incident Information Sharing Platfom. The exchange of Cyber Threat Information is handled by the TAXII servers, while the proposed Interledger solution offers data persistency. Furthermore, the SPC Layer leverages on the Universal Secure Gateway (USG) to provide secure information and data exchange. USG provides several Cyber Threat Intelligence services, including anomaly detection on the communication network, and exchanges such information with the components of the PHOENIX platform.

Furthermore, “by-design” measures to increase the resilience of the cyber-physical power system have been investigated. The proposed Resilience Enhancement Methodology aims to ensure the availability of automation functions required for the power system operation, based on the Double Virtualization (DV) approach. A use case related to PHOENIX LSP1 and, specifically, threat scenario ASM_SCADA_RTU, is defined and described in this deliverable. The Resilience Enhancement Methodology implementation includes two main parts, the functions and data layer, which is specific to the use case; and DV, which offers the framework for virtualization and distribution of the relevant functions. A laboratory set-up is defined to evaluate the Resilience Enhancement Methodology implementation. An initial evaluation scenario is defined and evaluation criteria are specified, including functionality and performance aspects. Finally, an outlook on planned improvements and refinements of the implementation is given.

This deliverable provides the second version of the SPC Layer description. The final update will be given in D2.4: Secure and Persistent Communications Layer (Ver. 2).

1. Introduction

The SPC layer is one of the key components to protect EPES assets and networks against cyber-attacks, enabling coordinated cybersecurity measures and the secure exchange of CTI. An initial description of the SPC Layer was given in D2.1: PHOENIX platform architecture specification [1] as part of the PHOENIX architecture description and was refined and expanded by the requirements of the SPC Layer in D2.2: Secure and Persistent Communications Layer (Ver. 0) [2]. This deliverable provides an update to D2.2.

The document is structured as follows:

This chapter provides a brief overview on the objectives of the PHOENIX project, the present deliverable and on the structure of the document. Chapter 2 describes the application of semantics and ontologies to the PHOENIX environment, specifically the TAXII 2.1 framework. The approach to transactions and communication is laid out in chapter 3. Chapter 3.2 describes the approach to federated ledger and Interledger in the SPC Layer, whereas chapter 5 specifies the initial communication platform. The Resilience Enhancement Methodology to implement resilience by design is presented in chapter 6 and results of the related simulation studies are included in chapter 7. The details of the PHOENIX implementation are included in deliverables describing the “PHOENIX integrated platform and SPaaS Core Services”, considering the classification level of this information. The initial version has been provided in D6.1: PHOENIX Integration guidelines and integrated platform (Ver. 0) [3] whereas an update to the PHOENIX architecture will be given in D6.2: PHOENIX Integration guidelines and integrated platform (Ver. 1).

2. Semantics and Ontologies – Application to the PHOENIX environment

This chapter describes the application of semantics and ontologies to the PHOENIX environment, which is built on the utilization of Trusted Automated Exchange of Intelligence Information (TAXII) v2.1 and Structured Threat Information eXpression (STIX). TAXII is designed to exchange CTI over HTTPS. It enables organizations to share CTI by defining an Application Programming Interface (API) that aligns with common sharing models. TAXII is specifically designed to support the exchange of STIX-formatted CTI data.

2.1. TAXII Server

SPC Layer is designed to support the secure exchange of different types of information among the various components. Thereby, a TAXII 2.1 server resides in the SPC Layer to exchange TAXII messages within the PHOENIX platform.

2.1.1. PHOENIX Pub/Sub Enhancements in TAXII Server

The TAXII protocol v2.1 considers the exchange of CTI information among TAXII Clients in a publish/subscribe mode via the TAXII "Channels" concept, as depicted in Figure 1. A TAXII Server maintains a TAXII Channel under an API root of the TAXII Server. For example, a single TAXII Server could host multiple API roots - one API root for Channels used by Sharing Group A and another API root for Collections and Channels used by Sharing Group B. Each API root contains a set of endpoints that a TAXII Client contacts in order to interact with the TAXII Server.

The protocol also specifies that TAXII Clients, which produce CTI, can publish messages to Channels and subscribe to Channels to receive published messages as CTI consumers. However, the protocol does not provide technical specifications of the TAXII Channels. The relevant section of the protocol specification [4] is empty and marked as "reserved" at the time of writing this deliverable.

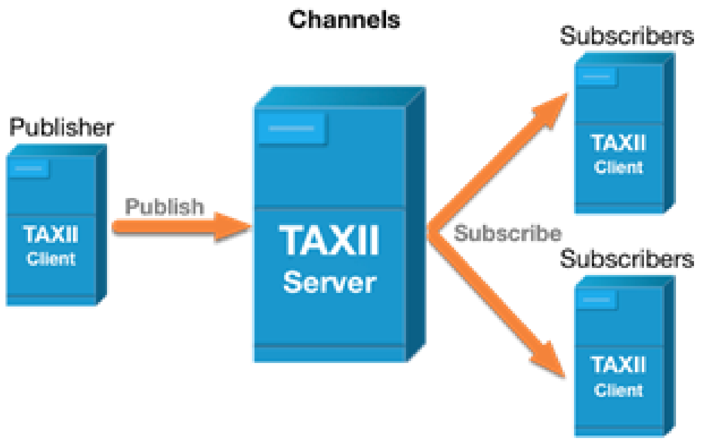

Figure 1: TAXII Channels for CTI exchange in a publish/subscribe manner [5]

In PHOENIX, the TAXII server will be based on Medallion [6], a minimal reference implementation of TAXII 2.1 Server in Python, which is an OASIS TC Open Repository, meaning that all contributions are subject to open source license terms expressed in the BSD-3-Clause License [7]. Medallion has been designed as a simple front-end REST server providing access to the endpoints defined in TAXII 2.1 specification. However, it does not incorporate any publish/subscribe mechanism.

Nevertheless, the publish/subscribe mechanism is critical for the PHOENIX TAXII Server, which will undertake the task of handling CTI exchange from PHOENIX components generating CTI towards the DLT of the SPC Layer. Thereby, it can establish secure federated communications among Large-Scale Pilot level PHOENIX components and between those components and Incident Information Sharing Platform (I2SP). Furthermore, this publish/subscribe mechanism perfectly suits the real-time notification objective of the PHOENIX project as it allows instantaneous, push-based delivery, eliminating the need for message consumers to periodically "poll" for new information and updates. This promotes faster response time and reduces the delivery latency that can be particularly problematic in critical energy systems where delays cannot be tolerated

To support this communication, the PHOENIX Consortium has developed a new middleware with interfaces for supporting multiple types of publish/subscribe or streaming technologies. Specifically, reference implementation is available for RabbitMQ [8], whereas Apache Kafka [9] is also planned to be added. Then, using the adopted PHOENIX channels approach, all PHOENIX components can subscribe to relevant topics such as Events, Attacks and Mitigations, and thus get notified on their desired topics, following the flow depicted in Figure 2.

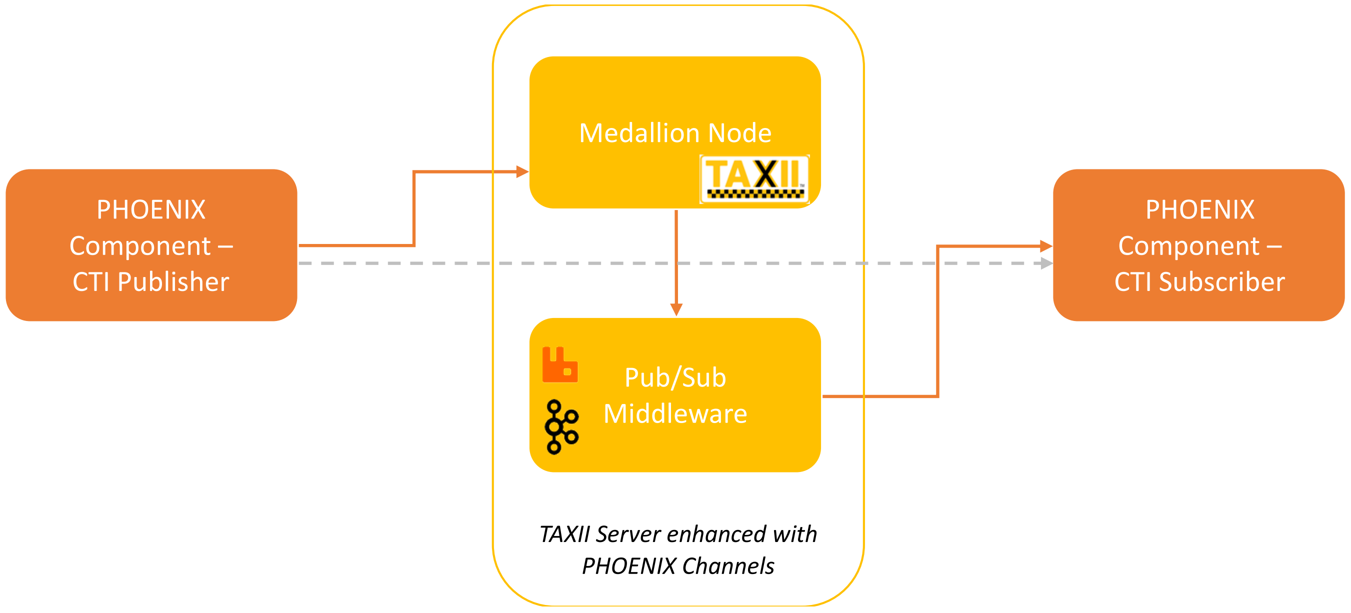
Medallion Node
TAXII
PHOENIX Component – CTI Publisher
PHOENIX Component – CTI Subscriber
Pub/Sub Middleware
TAXII Server enhanced with PHOENIX Channels

Figure 2: The publish/subscribe exchange mechanism in PHOENIX.

It must be noted that the TAXII server will also handle the authentication, authorization, and policy-based message handling, backed by the Keycloak server, implemented in the context of Accountability & Access Management (AAM) service of PHOENIX and detailed in deliverable D6.1: PHOENIX Integration guidelines and integrated platform (Ver. 0) [3].

2.1.2. TAXII Server in the SPC Layer

The SPC Layer, as it is reflected in its name, is designed to persist CTI data that is exchanged between different PHOENIX components. As stated in the previous paragraphs, the TAXII server handles these persistence operations by storing STIX data and metadata in its backend (normally a MongoDB instance [10]). However, this mechanism does not guarantee the correctness of STIX records maintained by TAXII server. To address this need, SPC utilizes DLTs, which are widely known as immutable data storages. DLTs enable the SPC Layer to assure the integrity of CTI data stored in the TAXII backend. To this end, the TAXII server follows the below procedure which is also presented in Figure 3.

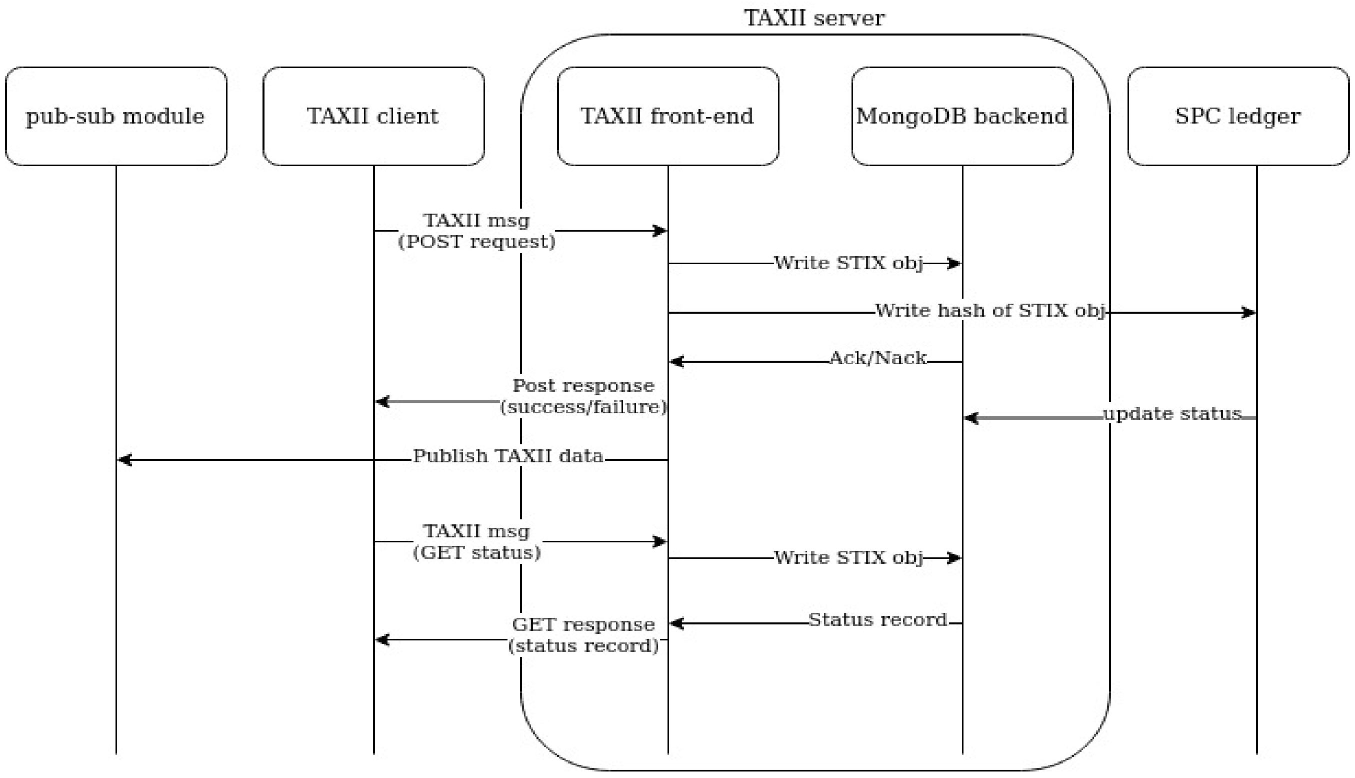

Figure 3: Sequence diagram for the write operation of TAXII data on TAXII server and SPC ledger.

1. The TAXII server's front-end upon reception of a POST request which is formatted in a TAXII message and carries STIX objects such as detected attacks generated by a producer, writes the TAXII data and metadata in the MongoDB backend.
2. At the same time, it triggers writing the hash of every STIX object which was encoded in the TAXII message in SPC's DLTs with the purpose of the data integrity enhancement. To achieve this, ledger-specific adapters are added to the basic TAXII server. These adapters allow the TAXII server to trigger transactions on different types of ledger. More precisely, ledger-specific adapters provide interfaces to the pre-defined smart contracts on SPC's DLTs. Consequently, the TAXII server can simply invoke functions on the target ledger. For the time being, the TAXII server includes adapters only for ConsenSys Quorum [11] and Hyperledger Fabric [12] ledgers, however, supports for other ledger types can be easily added by following the same principles. As stated above, only the hash of STIX objects are stored in ledgers. The rationale behind this decision is that the STIX objects generated by PHOENIX components might contain explicit or implicit personal data. Storing them in ledgers would lead to violation of the General Data Protection

Regulation (GDPR) right to be forgotten due to the immutable nature of DLTs. Thereby, the TAXII server utilizes MD5 hash function, which is a one-way function, and in practice infeasible to invert for generating hash of STIX objects.

3. After completion of the write operation on database, the MongoDB backend returns the operation status to the TAXII front-end.
4. The TAXII front-end creates a status resource to report back the request status provided by the backend to the client. The status resource contains the status of every STIX objects within the POST request and specifies whether the writing operation related to each object is still pending, successfully completed or failed. Next, the TAXII front-end persists the status resource in the MongoDB backend.
5. Since the duration of writing data on ledgers are highly variable and depends upon several factors such as the network condition, the TAXII front-end replies the POST request immediately after storing the status of the write operation in the backend. It is noteworthy that during this phase of the operation, the TAXII server sets the status of each individual object to either pending in case it is successfully written to the backend, or failure if the write operation has failed for any reason.
6. The TAXII v2.1 does not define the methods and data formats for the publish-subscribe communication despite the fact it has been promised in the standard specification. Therefore, the PHOENIX platform develops a separate module (i.e., pub-sub module) to fulfil this requirement which is thoroughly described in the following section. The TAXII server after sending the POST response, passes the TAXII data to the pub-sub module.
7. After finalizing publishing data on the SPC's ledger, the status of STIX object in the backend is modified accordingly. More accurately, the status is set to either success if the write operation in the ledger was successful or failure in case of unsuccessful attempt to write the data.

8. The TAXII client can later inquiry about the status of each STIX object using a separate TAXII message (i.e. GET status request). In this manner, the client can verify that the TAXII data and related metadata is successfully written in both MongoDB and SPC's ledger.

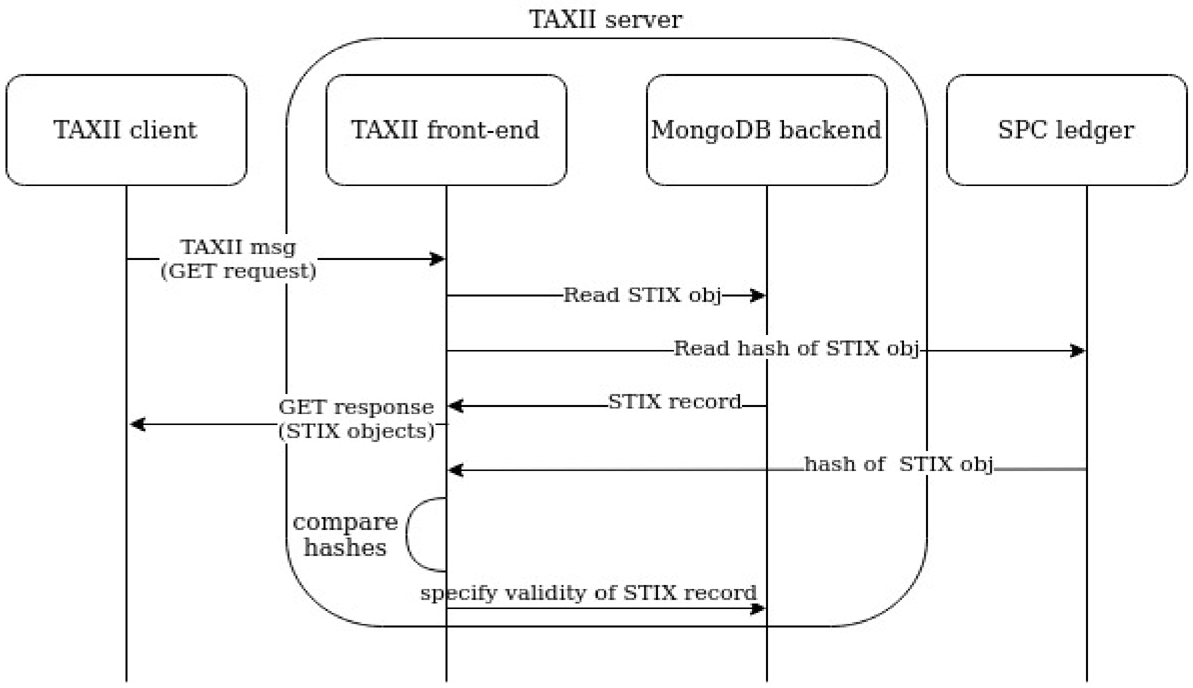

Figure 4: Sequence diagram for reading TAXII data from the TAXII server and SPC ledger.

In the PHOENIX platform, the majority of PHOENIX components receive the published CTI data through the channels dedicated to different types of STIX object. However, some components such as the Security Control Centre (SCC) dashboard follow the request-response communication paradigm for obtaining CTI data. To this end, they send a TAXII message carrying a GET request directly to the TAXII server. For instance, the SCC dashboard first collects IDs of the published STIX objects from the Configuration Maintenance Service (CMS) and then queries the related objects from the TAXII server. To handle GET requests, the TAXII server takes following steps as illustrated in Figure 4:

1. Upon reception of a GET request which is encapsulated in a TAXII message, the TAXII front-end retrieves the queried STIX objects from the MongoDB backend.
2. Simultaneously, the TAXII front-end triggers reading of the hash of the related STIX object from SPC's ledger through the corresponding ledger-specific adapter.
3. Like the write operation, the waiting time for completing the read operation in DLTs is variable. Hence, the TAXII front-end immediately replies to the GET request when the backend sends back the queried STIX objects.
4. When the front-end receives the hash of the object from SPC's ledger, it compares it with the hash of the actual STIX object provided by the backend. The identical hashes imply that the STIX record on the backend is valid while any discrepancy in the hashes indicates some alteration in the data record. In the latter case, the front-end notifies the backend about the discrepancy and subsequently the backend labels the STIX record as an invalid entry. It is noteworthy that the front-end rejects any GET request for the invalid STIX object. Thereby, this procedure can detect

any malicious behaviours aimed at modifying STIX records in the MongoDB backend, which results in a higher degree of data integrity.

2.1.3. Authentication on TAXII Server

As already mentioned, authorization in the TAXII server uses Keycloak, a central Authorization, Authentication, Accounting (AAA) OAuth2 service. To access the server for read or write operations, the user must have access to an account with access to the "taxii-read" and "taxii-write" scopes respectively. The TAXII server can be configured on deployment to use the correct Keycloak instance and details (URL, client id, client secret, realm). When making an HTTP request to the server, the client must provide authorization credentials in one of two ways:

1. Using a valid JSON web token obtained independently from the Keycloak instance, that provides access to the correct scopes as mentioned above. The token is sent in the “Authorization” header with the prefix “Token” and validated in the TAXII server.
2. Using basic authentication, as per the TAXII protocol. In this case, the TAXII server uses the credentials to obtain authorization from Keycloak on behalf of the client, using the OAuth2 password flow. Basic authorization is defined in RFC 7617 [13].

2.1.4. Ledger-specific adapter for TAXII Server

As described in section 2.1, the TAXII server communicates with SPC’s ledgers via ledger-specific adapters. These adapters provide interfaces toward different ledger types. To this end, each adapter first setups a connection between the TAXII server and corresponding ledger type. After that, for the write operation, it triggers a transaction by invoking the pre-defined function (i.e., emitData) from the configured smart contracts on the ledger. Every function call on an SPC ledger, which also carries the hash of the STIX object as an input, is persisted in the ledger. For the read operation, the adapter uses the transaction IDs to retrieve the STIX object hashes from the ledger, which will be further examined against the hashes of the STIX records in the TAXII backend.

2.2. TAXII Client

Many components within the PHOENIX platform, such as Situation Awareness, Perception & Comprehension (SAPC) and Incident Mitigation & Enforcement Countermeasures (IMEC), produce substantial amount of CTI data, which need to be shared with other PHOENIX services and components. To achieve this goal, the PHOENIX components and services regardless of their roles, which might be either CTI producer, consumer, or both, must feature a TAXII client that is a minimal client implementation for the TAXII 2.1 server specification. In the PHOENIX setup, the TAXII server is mainly utilized for CTI data exchange between clients. For this purpose, it maintains a repository of CTI objects provided by CTI producers and replies to consumer queries. The TAXII 2.1 server implementation and related functionalities are extensively discussed in the following section.

3. Transaction and Communication

This chapter describes the approach to deployment of the PHOENIX components based on the federated cloud and investigates Cloud Native Applications for 5G communications.

3.1. Cloud Native for 5G communications

In this section, we provide a background around the Cloud Native Applications, microservices, service mesh, with modelling 5G cloud-native applications by exploiting the service mesh paradigm. The PHOENIX project can benefit from the service mesh approach to strengthen its security by design. In the current document, we give a general overview of the service mesh, and we will provide more detailed implementation specific to PHOENIX in the next deliverable.

3.1.1. Cloud Native Paradigm

Referring to the definition of Cloud Native Application (CNA) in [14], it is a distributed, elastic, and horizontal scalable system composed of (micro) services, which isolates states in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform.

Based on this definition, we provide the description of the main concepts proposed or defined by standardizing initiatives or other research works:

- Elasticity has been defined by [15] as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.
- Scalability [16] can be differentiated to structural scalability and load scalability. Structural scalability refers to the ability of a system to expand in a chosen dimension without major modifications to its architecture. Load scalability is the ability of a system to perform gracefully as the offered traffic increases.
- A microservices-oriented architecture represents a way to develop a single application as an amalgamation of small, independent services, each running in its own process and communicating with lightweight mechanisms, often a Hypertext Transfer Protocol (HTTP) resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery [17]. Another description of the microservice is that it has two functions: 1) Business Logic, which implements the business functionalities, computations, and service composition/integration logic, and 2) Network Functions, which take care of the inter-service communication mechanisms (e.g., basic service invocation through a given protocol, apply resiliency and stability patterns, service discovery, etc.).
- A self-contained deployment unit is described as a part of the application’s deployment topology for realizing a specific technical unit [18]. More and more often, a deployment unit is understood as a standard container. A standard container encapsulates a software component and all its dependencies in a format that is portable, so that any compliant runtime can run it without extra dependencies, regardless of the underlying machine and the contents of the container.

- Stateful components are used for multiple instances of a scaled-out application component to synchronize their internal state to provide a unified behaviour [19].
- Elastic platform is understood as a middleware for the execution of custom applications, their communication, and data storage is offered via a self-service interface over a network [19].

Another related concept for managing microservices are the service mesh services. We give a brief description, with its main components, and example of research based on 5G-ready applications.

3.1.2. Service Mesh

A Service Mesh [20] is a dedicated infrastructure layer with a set of deployed infrastructure functions that facilitate service-to-service communication through service discovery, routing and internal load balancing, traffic configuration, encryption, authentication, authorization, metrics, and monitoring. It provides the capability to define network behaviour, microservice instance identity, and traffic flow through policy in an environment of changing network topology due to service instances coming and going offline and continuously being relocated [21].

The Service Mesh can be defined also as a distributed computing middleware that optimizes communications between application services. The service-to-service communication is enabled using a proxy. A Service Mesh is implemented as an array of lightweight network proxies that are deployed alongside application code. In addition, the Service Mesh can be leveraged to monitor and secure communication: The Service Mesh can learn and smartly route traffic.

3.1.3. Service Mesh Components & Capabilities

A Service Mesh consists of two main architectural layers or components [22]:

- Data plane: The interconnected set of proxies in a Service Mesh that control the inter-services communication represents its data plane. The data plane is the data path and provides the ability to forward requests from the applications. The specialized proxy that is created for each service instance (i.e., sidecar proxy) performs the runtime operations needed for enforcing security (e.g., access control, communication-related), which are enabled by injecting policies (e.g., access control policies) into the proxy from the control plane. A data plane may provide more sophisticated features like load balancing, authentication, and authorization.
- Control plane: A control plane is a set of APIs and tools used to control and configure data plane (proxy) behaviour across the mesh. The control plane is where users specify authentication policies and naming information. The intelligence and data required for implementing all security functions lie in the control plane. These include the software for generating authentication certificates and the repository for storing those, policies for authentication, authorization engine, software for receiving monitoring data regarding each microservice and aggregating them.

As part of the process of providing the communication, the following capabilities are supported:

- Secure communication – Mutual Transport Layer Security (TLS), encryption, dynamic route generation, multiple protocol support, including protocol translation where required (e.g., HTTP1.x, HTTP2, gRPC, etc.).

- Authentication and authorization – Certificate generation, API keys, key management, whitelist, and blacklist, Single Sign-On (SSO) tokens.
- Secure service discovery – Discovery of service endpoints through a dedicated service registry
- Resilience/stability features for communication – fault injection/handling, load balancing, failover, rate limiting, request shadowing.
- Observability/monitoring features – Logging, metrics, distributed tracing.

Ingress Controller

The service proxy of a Service Mesh can be deployed for control of ingress traffic (i.e., external traffic coming into microservices application as opposed to microservice-to-microservice communication). It realizes the functions of an API gateway. The main ingress controller functionalities are:

- A common API for all clients shielding the actual API inside the Service Mesh.
- Composition of results received from calls to multiple services inside the Service Mesh in response to a single call from the client.
- Load balancing.
- Public TLS termination.

Egress Controller

The service proxy of a Service Mesh can be deployed for control of egress traffic (i.e., internal traffic coming from microservices destined for microservices outside of the mesh). Conceptually, the egress controller can be looked upon as a sidecar proxy for one or more external servers.

The egress proxy provides the following functions:

- Protocol translation from microservice-friendly protocols (e.g., RPC/gRPC/REST) to web-friendly protocols (e.g., HTTP/HTTPS).
- Credential exchange: Translate from internal (mesh) identity credentials to external credentials (e.g., SSO tokens or API keys) without the application directly accessing the external system's credentials.
- A single set of workloads (e.g., hosts, IP addresses) to whitelist for communication to external networks (e.g., firewalls can be configured to allow only egress proxies to forward traffic out of the local network).

Various research and industrial efforts have emerged to deploy 5G-ready application with service mesh. In particular, in [23] the authors propose a 5G-ready application consisting of cloud-native components that rely on a service mesh infrastructure as a mean of network abstraction. The service mesh operates on top of a programmable 5G environment. To exploit the service mesh added-value, a 5G full-stack architecture must be used in [24] , which relies on a solid interplay between various logical layers such as the actual data plane, the service mesh control plane, and the configured virtualized resources that are offered by the telco provider as a proper slice.

3.2. Deployment based on the (federated) cloud

In an attempt to increase the availability and reliability of the PHOENIX core components and foster communications resilience, the platform deployment of PHOENIX follows the cloud native paradigm with the adoption of container-based operation (all components operate in the form of Docker containers) and relevant container orchestration processes, Kubernetes in particular (see [25], [26] and [3] for details and discussion). Figure 5 highlights the Kubernetes-based deployment architecture adopted to support the component deployments.

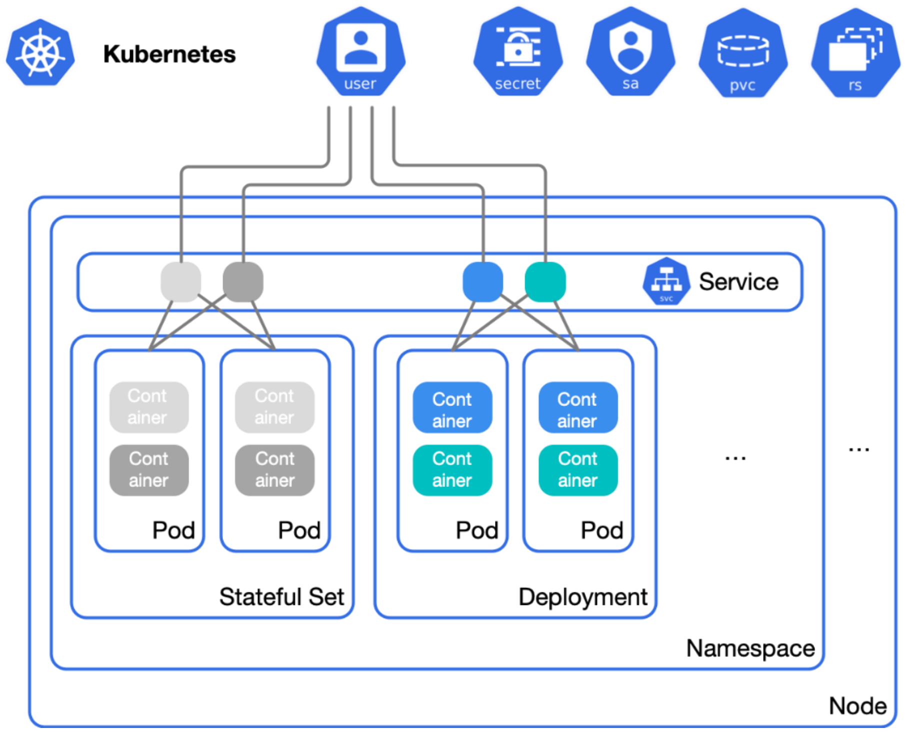

Figure 5: Generic Kubernetes concepts architecture used in PHOENIX.

Indeed, we consider that a PHOENIX cluster consists of several nodes, hosting a number of namespaces; typically, each set of components (SPC, SAPC, Privacy Protection Enforcement (PPE) etc) belongs to the same *namespace* and consisting of a set of services, bundled under the scope of Kubernetes *deployments* or *stateful sets* (see [25] for details on the specific Kubernetes resources definitions). Accordingly, the services are typically broken down in *pods*, each pod holding a running (Docker) container. Figure 6, below, showcases the deployment and contained pods relevant to the SPC DLT plane whereas Figure 7 depicts the pods relevant to the PHOENIX-adapted TAXII server.

```
$ kubectl -n spc get deployments
NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
quorum-node1-deployment    1/1     1            1           31d
quorum-node2-deployment    1/1     1            1           31d
quorum-node3-deployment    1/1     1            1           31d
```

```
$ kubectl -n spc get pods
NAME                                      READY   STATUS             RESTARTS   AGE
quorum-node1-deployment-6ff7f75546-n9njk   2/2     Running            1          7d1h
quorum-node2-deployment-6cfd544b44-kj7tl   2/2     Running            1          102m
quorum-node3-deployment-64f7fd97b9-pqt8h   2/2     Running            1          7d1h
```

Figure 6: Structure of the SPC DLT components (deployments and pods) under the SPC namespace of Kubernetes.

```
$ kubectl -n phoenix get pods
NAME                                 READY   STATUS    RESTARTS   AGE
medallion-f4f745f59-87fv7            1/1     Running   0          7d7h
medallion-worker-5d7f458cb7-zfr9c    1/1     Running   10         10d
mongodb-9bcd68959-97vsv              1/1     Running   2          32d
rabbitmq-0                           1/1     Running   2          67d
```

Figure 7: Structure of the Kubernetes pods regarding the TAXII-related components.

It is worth mentioning that, apart from the standard methods for increasing availability (e.g. with the introduction of replica sets – an approach readily adopted by the project Kubernetes deployments) and in order to further foster the increased availability perspective of PHOENIX at the level of both service and data provisioning, cloud-native storage solutions have also been sought with the adoption of the Rook framework [27] [28] in combination with the CEPH storage backend [29], as shown in Figure 8.

```
$ kubectl -n rook-ceph get pods
NAME                                            READY   STATUS      RESTARTS   AGE
csi-cephfsplugin-provisioner-5c65b94c8d-bn988    6/6     Running     83         165d
csi-cephfsplugin-vklr5                          3/3     Running     30         165d
csi-rbdplugin-provisioner-569c75558-k4t6c        6/6     Running     82         165d
csi-rbdplugin-qq5kc                             3/3     Running     30         165d
rook-ceph-mgr-a-55467b4f7-87tx7                 1/1     Running     9          165d
rook-ceph-mon-a-77dd5dfb66-lcljk                1/1     Running     9          165d
rook-ceph-operator-69d45cb679-5fjcd             1/1     Running     9          165d
rook-ceph-osd-0-57cd6975cb-5cqm4                1/1     Running     9          165d
rook-ceph-osd-prepare-compute-r540-1-7zp6l       0/1     Completed   0          7d9h
rook-discover-7ntnq                             1/1     Running     9          165d
```

Figure 8: Kubernetes pods governing the cloud-storage configuration of PHOENIX.

4. Federated ledger & Interledger on SPC Layer

This chapter provides an overview of DLT and Interledger solutions and the solution adopted in PHOENIX for interoperability of different ledgers as a persistent and immutable data storage.

4.1. Distributed Ledger Technologies and Interledger solutions

Many application scenarios utilize DLTs, mainly as immutable data storages [30]. However, the immutability of data highly depends upon the consensus mechanisms used by a DLT. For instance, public ledgers often use Proof-of-Work (PoW) as the consensus algorithm (e.g., Bitcoin [31]) which makes modification of the stored data substantially hard, at the expense, though, of longer processing time and higher energy consumption. On the contrary, private permissioned ledgers commonly utilize less complex consensus mechanisms such as Proof-of-Authority (PoA), a practice that results in lower level of trust [32]. In recent years, several Interledger solutions have been proposed to combine the strengths of different ledgers while combating their shortcomings at the same time.

In [33], authors studied a wide range of Interledger approaches and eventually categorized them into six groups based on various metrics including value transfer/exchange, cost, trustworthiness, scalability, and privacy. These categories are as follows:

1. atomic cross-chain transactions, mainly devised for exchanging digital assets among ledgers [34],
2. transactions across a network of payment channels which performs off-chain trading of assets [35],
3. the W3C Interledger Protocol (ILP) which provides a more generic form of the previous category for transferring funds among DTLs [36],
4. bridging which facilitates bidirectional data/value transfer among DLTs [37],
5. sidechains that are designed to enhance the overall throughput and trust of Interledger communication by moving some of the transactions to other ledgers (so called sidechains) [38], and
6. ledger-of-ledgers, which utilizes a designated ledger to harmonize multiple sidechains [39].

The Interledger solutions benefit from hash-locks to guarantee the atomicity of cross-ledger operations. To this end, cryptographic locks are utilized to lock digital assets. On the other hand, for unlocking the assets, a secret that is used to generate hash-locks in first place, needs to be revealed. Consequently, this mechanism ensures that all transactions on all involved ledgers either succeed or fail. However, it is possible that a secret may fail to be revealed and thereby assets remain locked. To counteract this problem, hash-locks are typically accompanied with time-locks (i.e., also known as Hashed Time Locked Contracts (HTLCs)) which allow the locked assets to be released after a certain time interval [40].

4.2. Interledger solution

The SPC Layer adopts a data-centric approach based on federated DLTs to achieve a higher degree of persistency, traceability, availability, integrity, and interoperability in the context of data communications. As stated above, different ledger technologies meet various needs. For instance, the ledgers aimed for asset and access control management are preferably chosen from permissioned ledgers with the purpose of achieving cost reduction whereas public ledgers are widely used for trade and making payments due to their high level of trustworthiness. Therefore, beyond usage of DLTs within SPC Layer, different components of the PHOENIX platform can also leverage the use of DLT as a persistent and immutable data storage.

Concerning utilization of multiple ledgers simultaneously within the PHOENIX platform, it is imperative to develop a distributed and resilient solution for interconnecting varying ledger networks, which are also suitable for critical EPES assets and data. For enabling interactions among differing DLTs, several Interledger approaches, that some of them are discussed in the previous section, were developed [33] [41]. The SPC Layer benefits from the Interledger (IL) component, which is implemented on basis of protocol bridging. More precisely, the IL component triggers transactions on multiple ledgers (so-called Responder ledgers) when it receives an event from one ledger (so-called Initiator ledger). It is noteworthy that all cross-ledger operations performed by IL are atomic. The implementation of the PHOENIX IL component, which is built on top of the SOFIE IL module [42, 43, 44, 45] enables connection between various DLTs including Quorum, Hyperledger Fabric, Hyperledger Indy, Ethereum and KSI. However, IL can be easily extended to support other ledger networks.

The inter-ledger component facilitates the integration of multiple ledgers, which results in cohesive storage platforms where different types of the ledgers can be used simultaneously to benefit from the strengths of every ledger and overcome its downsides. For better understanding, let us consider the I2SP component in PHOENIX platform, which is aimed to bridge the gap between national and global Computer Emergency Response Team (CERTs) and different EPES operators. I2SP would be instantiated as a distributed platform comprising one or more ledgers where each stakeholder can establish agreements and share CTI, such as cyber-attacks, with others reliably and transparently through the IL component.

In the PHOENIX platform, the IL component can be utilized for following application scenarios:

- Storing Data Hashes. Writing complete set of data records on a public ledger is relatively expensive and time-consuming process due to its demanding consensus mechanism. Thereby, typically the full data blocks are stored in a private ledger while a public ledger stores only a hash of the data to assure the higher level of trust.
- Transferring Data among different ledger types.
- Exchanging Digital assets. To achieve this, IL benefits from HTLCs to automate the process of trading value between DLTs.

Every PHOENIX component depending on its requirements, may utilize different ledger types. As discussed in deliverable D2.2 [2], information (e.g., CTI) generated within PHOENIX platform cannot be shared with everyone, especially ledger nodes outside the LSP premises. Therefore, only private permissioned ledgers such as Quorum and Hyperledger Fabric where only certain nodes are authorized to execute varying transactions on data blocks are used by PHOENIX components. As shown in Figure 9, each component (I2SP, TAXII server and Privacy Protection Enforcement (PPE)) can directly access corresponding DLT while the IL component enables connections to other ledgers. In other words, multi-ledger operations are only executed through IL.

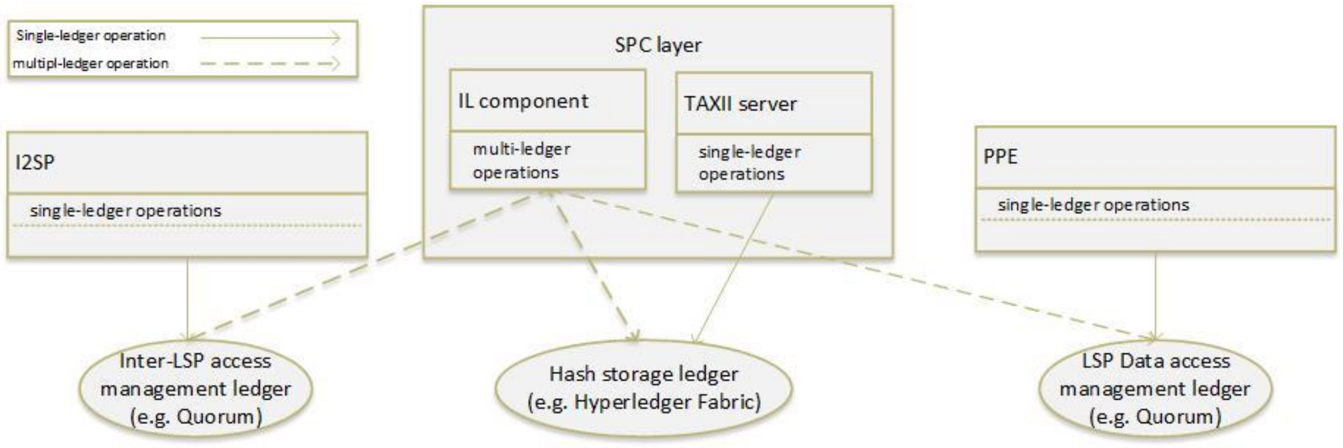

Figure 9: Relationship between PHOENIX components and DLTs.

4.2.1. Interfaces and Services

The IL component for every cross-ledger operation instantiates a unidirectional connection between an Initiator ledger and multiple Responder ledgers. Thereby, for enabling two-way communication between ledgers, IL requires two instances of unidirectional connection. It is worth mentioning that the current IL implementation supports exactly one Initiator ledger, however, it is plausible for a respective PHOENIX component to overcome this limitation through other approaches (e.g., smart contracts).

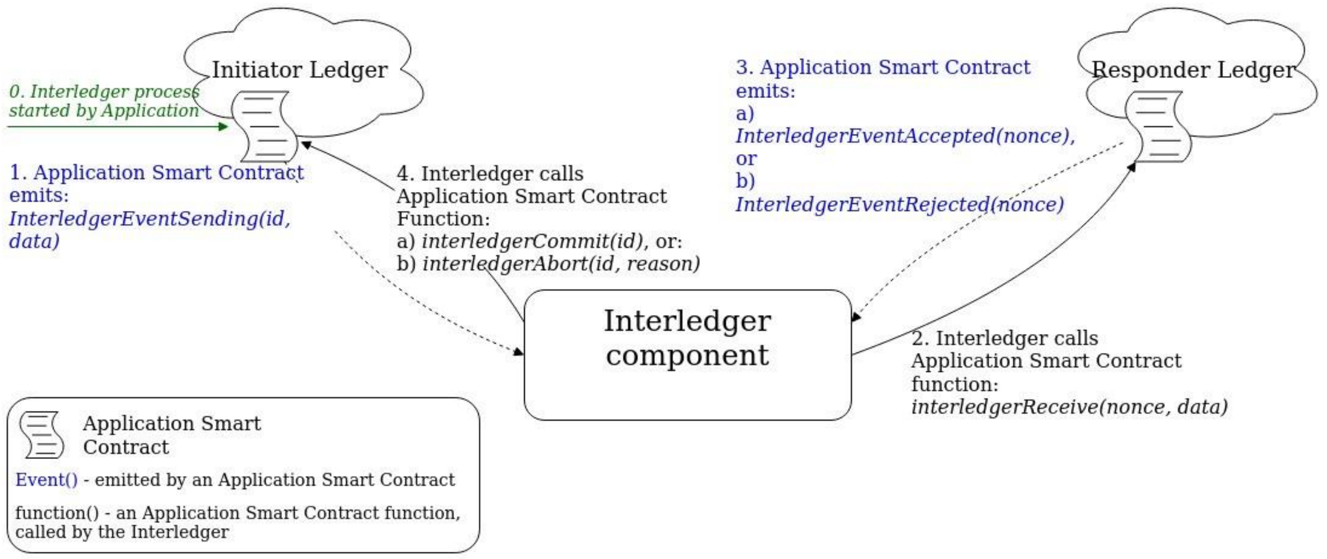
Initiator Ledger
0. Interledger process started by Application
1. Application Smart Contract emits: InterledgerEventSending(id, data)
4. Interledger calls Application Smart Contract Function: a) interledgerCommit(id), or: b) interledgerAbort(id, reason)
3. Application Smart Contract emits: a) InterledgerEventAccepted(nonce), or b) InterledgerEventRejected(nonce)
Responder Ledger
Interledger component
2. Interledger calls Application Smart Contract function: interledgerReceive(nonce, data)
Application Smart Contract
Event() - emitted by an Application Smart Contract
function() - an Application Smart Contract function, called by the Interledger

Figure 10: The Interledger component workflow [46].

Figure 10 demonstrates how the IL component implements the bridging protocol. Each PHOENIX component that is interested in executing an operation on other ledgers rather than respective DLT needs to deploy a specific Smart Contract (SC) that includes the Sender and Receiver interfaces. The interfaces would be chosen according to the role, which the component is playing throughout the operation (i.e., either Initiator, Responder, or both). Table 1 briefly describes the Sender and Receiver interfaces in the Initiator and the Responder ledger, respectively. Next, the IL component configures a connection between target ledgers and starts listening for *InterledgerEventSending* events from the configured Initiator ledger SC. When a specified event is emitted (step 1), the IL component triggers a transaction carrying the received information on the Responder ledger by invoking the *interledgerReceive* function (step 2). On the Responder ledger, the configured SC indicates the transaction status through emitting either an *InterledgerEventAccept* event or an *InterledgerEventReject* event (step 3). At the final stage (step 4), IL notifies the Initiator ledger SC about the status of the data transfer by calling relevant functions (i.e., *InterledgerCommit* in case of an *InterledgerEventAccept* event and *InterledgerAbort* upon reception of an *InterledgerEventReject* event).

Table 1: Interledger interfaces [46].

Interface Details	
Name	Sender Interface
Description	Used by the configured Initiator ledger SC to trigger cross-ledger operations
Input	Data aimed to transfer/value aimed to exchange
Output	Status of transaction (i.e., success/failure)
Name	Receiver Interface
Description	Used by the configured Responder ledger to process the cross-ledger operations
Input	Data aimed to transfer/value aimed to exchange
Output	Status of transaction (i.e., success/failure)

The *InterledgerEventSending* events emitted from the configured Initiator ledger SC apart from the actual data, carries an *id* variable. The IL component benefits from this *id* to internally map a certain event to the triggered transaction; however, IL does not share the *id* with the Responder ledger SC. Therefore, the *id* parameter is not necessarily unique and can be utilized to categorize a certain type of activities on the Initiator ledger while preserving the privacy of the Initiator ledger. In step 2, for every function invocation on the Responder ledger SC, IL generates a *nonce* that is used in later steps (i.e., step 4) to make a mapping between the transaction triggered on the Responder application and the event from the Initiator ledger. More accurately, this variable enables IL to send back the status of the transaction to the correct Initiator application. It is noteworthy that Figure 10 details the workflow of the IL component for Quorum ledger types, however, there may be slight differences in the overall workflow of IL based upon communicating ledger types. For instance, Hyperledger Fabric ledgers use chaincodes instead of smart contracts. Thereby, IL adds supports for different DLTs through ledger-specific adapters.

4.2.2. Internal Architecture

Figure 11 presents the overall architecture of the IL component. As shown in Figure 11, the IL component is divided into two main parts: 1) the ledger-specific adapters that allow certain types of DLT to run applications as either Initiator ledger, Responder ledger or both, and 2) the *IL Core*, which tracks the status of ongoing transactions and exchange transaction specific parameters between Initiator and Responder adapters. However, the adapters can be extended to support processing of the data variable transferring between Initiator and Responder applications, but the IL core has not envisioned to support any operation on the data. The current IL component implements Quorum and Hyperledger Fabric adapters for both Initiator and Responder roles.

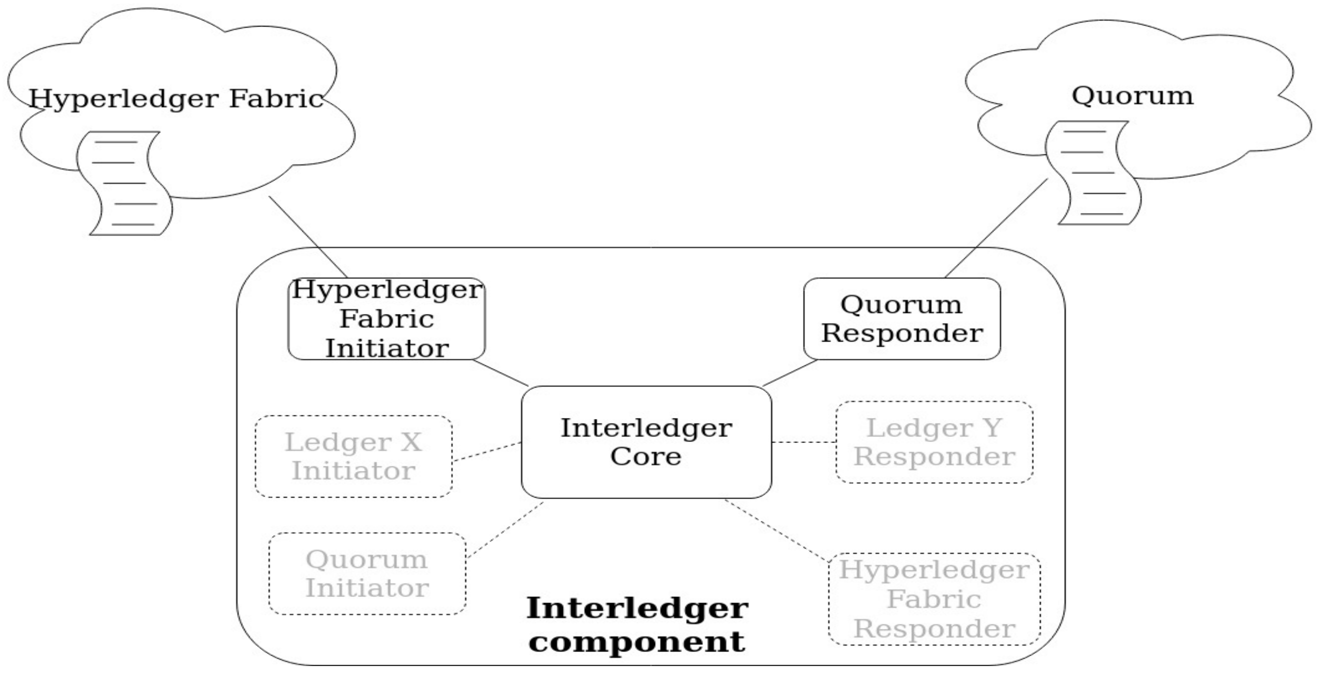

Figure 11: Interledger component architecture [46].

The IL component instantiates a unidirectional connection between ledgers in two modes: 1) *one-to-one mode* where one Initiator application communicates with only one Responder application, and 2) *multi-ledger mode* where one Initiator ledger connects with multiple Responder ledgers. The multi-ledger mode handles transactions in two ways: 1) all Responder applications need to validate the transaction or 2) *k* out of *N* Responders must verify the transaction; otherwise, the operation would be rejected.

Figure 12 describes how IL generally performs a cross-ledger operation in the *one-on-one mode.* To start this process, the *IL core* invokes the *listen_for_events* function on the Initiator adapter (step 1) where the *InterledgerEventSending* events from an Initiator application are being filtered (step 2). Upon reception of the event, the Initiator adapter invokes the *send_transfer* function from the *IL core* (step 3). This function generates a random number (i.e., called nonce) which is used to map the original *id* parameter to the ongoing transaction. Next, it passes the nonce and the data parameter to the Responder adapter by calling the *send_data* function (step 4). Additionally, at this stage of the operation, the IL core creates a state object where all transaction-specific information would be stored. As a following step, the Responder adapter invokes the *interledgerReceive function* from the Responder application (step 5). On the Responder ledger, the status of the transaction is sent back via either the *InterledgerEventAccepted* event or *InterledgerEventRejected* event (step 6). The Responder adapter then calls the *process_result* function (step 7) for transferring the transaction status to the *IL core* where the nonce is mapped to the id parameter used by the Initiator application. After that, the *IL core* depending on the transaction status invokes either *commit_sending* or *abort_sending* function from the Initiator adapter (step 8). The whole operation would be concluded by invoking either the *InterledgerCommit* or *InterledgerAbort* from the Initiator application (step 9).

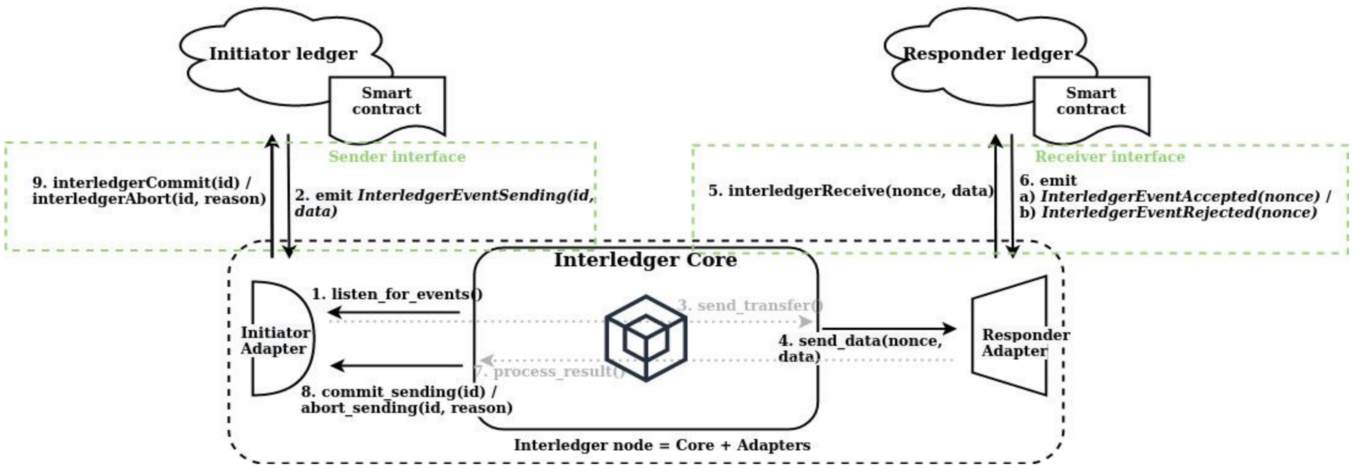
Initiator ledger
Smart contract
Responder ledger
Smart contract
Sender interface
Receiver interface
9. interledgerCommit(id) / interledgerAbort(id, reason)
2. emit InterledgerEventSending(id, data)
5. interledgerReceive(nonce, data)
6. emit a) InterledgerEventAccepted(nonce) / b) InterledgerEventRejected(nonce)
Interledger Core
1. listen_for_events()
3. send_transfer()
4. send_data(nonce, data)
7. process_result()
8. commit_sending(id) / abort_sending(id, reason)
Initiator Adapter
Responder Adapter
Interledger node = Core + Adapters

Figure 12: One-to-one mode of the Interledger component [46].

As shown in Figure 13, for the multi-ledger mode the steps related to the Initiator pipeline (i.e., 1-2 and 12-13) are identical to the one-to-one mode, however, the Responder pipeline is handled differently in this mode. In detail, the multi-ledger mode employs a *two-phase commit* approach where the transaction would be only committed if a certain number of Responder applications accepts the transaction. To this end, the *IL core* invokes the *send_data_inquire* function on all Responder adapters (step 4) which results in calling the *interledgerInquire* function of respective Responder ledger SCs (step 5). Upon reception of replies from Responder applications, which would be in form of either an *InterledgerInquiryAccepted* or an *InterledgerInquiryRejected* event (step 6), the adapters invoke the core's *transfer_inquiry* function (step 7). Once a sufficient number of Responder adapters delivers the replies to the *IL core*, it will then call either *send_data or abort_send_data* function for committing or aborting the transaction respectively (step 8).

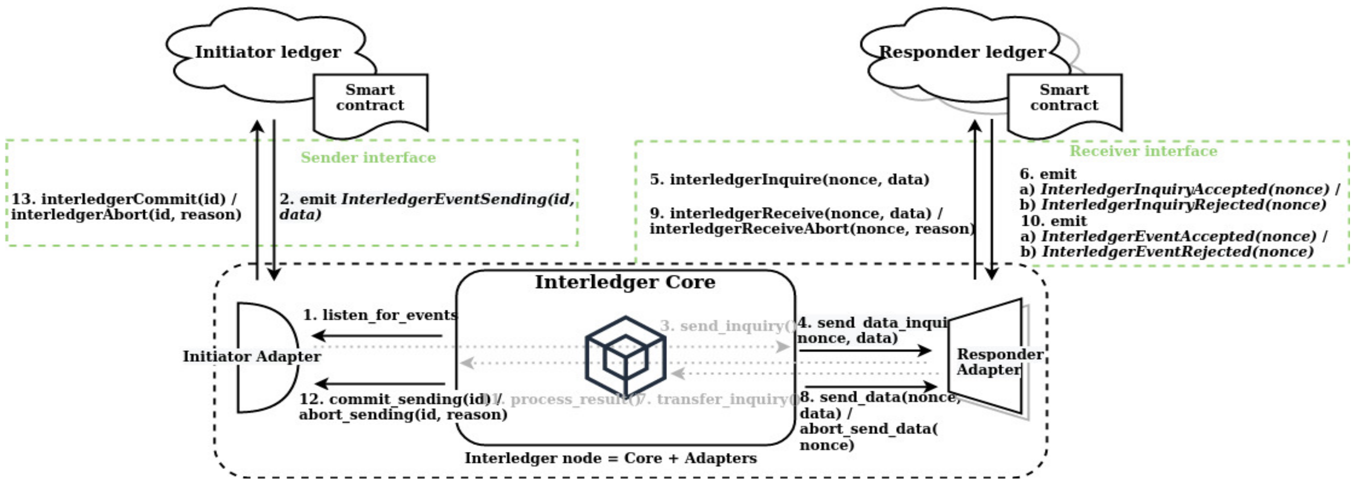
Initiator ledger
Smart contract
Responder ledger
Smart contract
Sender interface
13. interledgerCommit(id) / interledgerAbort(id, reason)
2. emit InterledgerEventSending(id, data)
5. interledgerInquire(nonce, data)
9. interledgerReceive(nonce, data) / interledgerReceiveAbort(nonce, reason)
Receiver interface
6. emit a) InterledgerInquiryAccepted(nonce) / b) InterledgerInquiryRejected(nonce)
10. emit a) InterledgerEventAccepted(nonce) / b) InterledgerEventRejected(nonce)
Interledger Core
1. listen_for_events
Initiator Adapter
3. send_inquiry()
4. send_data_inquiry(nonce, data)
Responder Adapter
12. commit_sending(id) / abort_sending(id, reason)
11. process_result()
7. transfer_inquiry()
8. send_data(nonce, data) / abort_send_data(nonce)
Interledger node = Core + Adapters

Figure 13: Multi-ledger mode of Interledger component [46].

These function calls result in invocation of either the *InterledgerReceive* or the *InterledgerReceiveAbort* function on the Responder application SCs (step 9)*.* The Responder ledgers report the status of the transactions to the adapters via either *InterledgerEventAccepted* or *InterledgerEventRejected* events. Next, the adapters forward the results to the *IL core* through the *process_result* function calls (step 11). Finally, the transaction is concluded on the Initiator side in a similar manner to the one-to-one mode. It is worthy to mention that the current IL component only implement the one-to-one mode, however, the implementation of the multi-ledger is in progress.

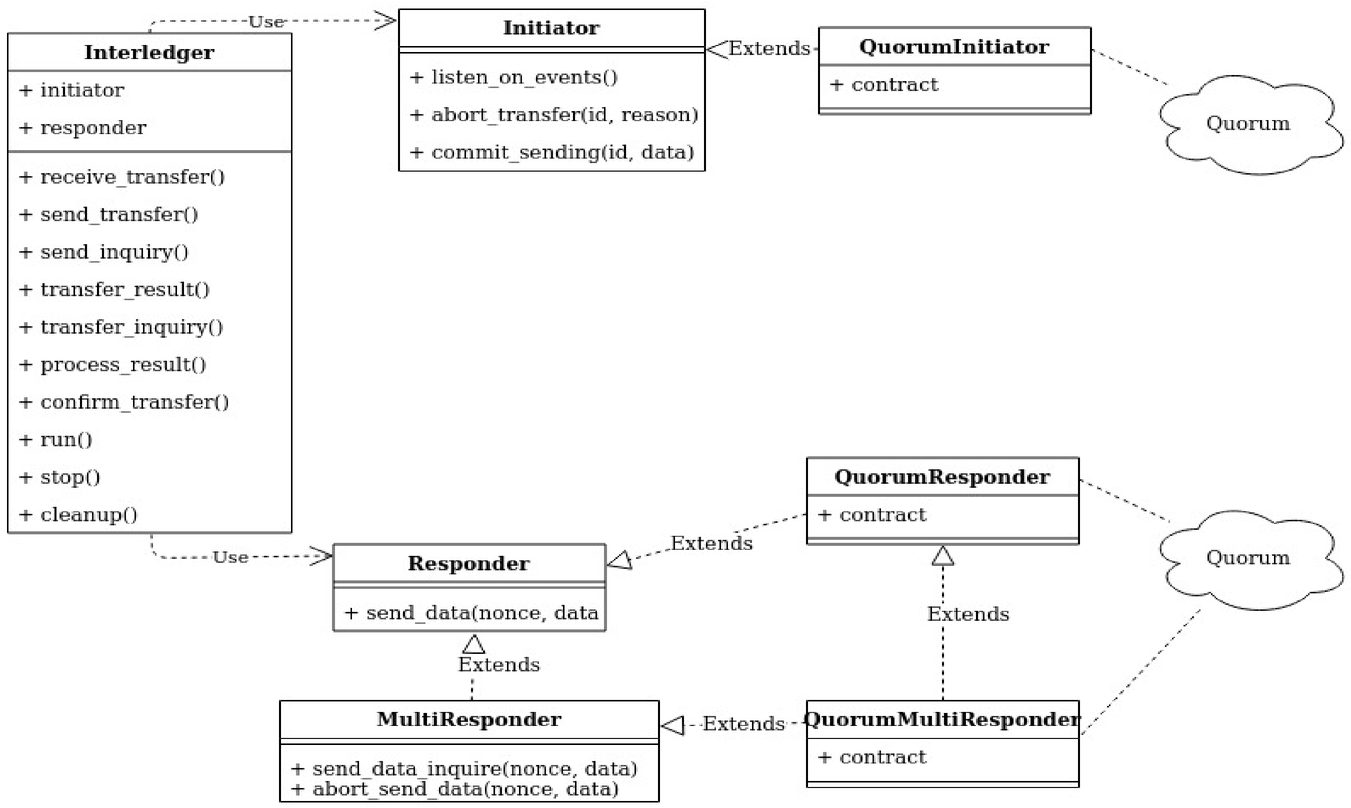

Figure 14: The class diagram of the IL component [46].

Figure 14 presents the class diagram of the IL component. As stated before, IL supports different ledger types; therefore, it defines abstract APIs that later could be implemented for each type of ledger. For better understanding, the Quorum Initiator and Responder classes are demonstrated as an example in Figure 14.

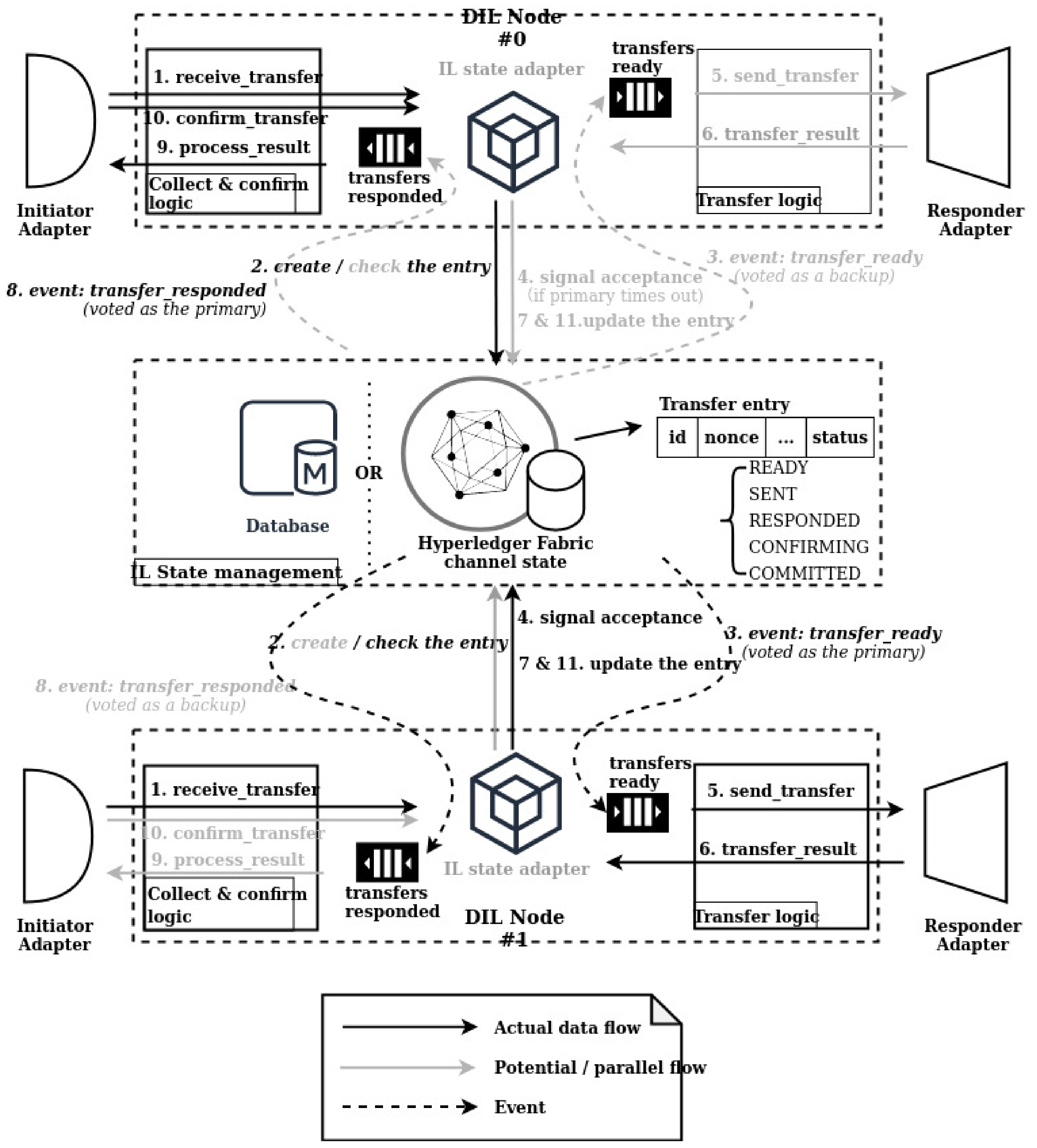

Figure 15: Architecture of Decentralised Interledger [46].

A Decentralised Interledger (DIL) approach where a number of nodes are simultaneously offering Interledger services is envisioned for future releases. In the DIL design, every party runs one or more IL component where all members share the identical state information. Consequently, in this approach the applications and their users must trust the group of IL nodes rather than a single node, which also leads to higher degree of reliability and availability. Despite several changes required by the DIL solution to the internal structure of the IL component, the IL interfaces, which are currently used by the applications, would remain the same.

As Figure 15 depicts the fact that the DIL architecture shares a lot of similarities with basic IL (i.e., run on a single node). The necessary functionalities for DIL solution will be implemented in the *IL core* while *the ledger-specific adapters* and application interfaces will remain intact. Unlike the basic IL where the *IL core* on a single node is responsible for forwarding all transactions to the target *ledger-specific adapters*, in the DIL architecture a consortium of nodes is handling all transactions. To this end, the *IL core* of every node stores the related transaction-specific information into the shared state, which is accessible by all nodes. Next, using a deterministic algorithm a pseudo-random node from the consortium is selected to proceed with the transaction in parallel with the standby nodes, which provide higher degree of redundancy by tracking the target ledgers.

Figure 15 in addition to the DIL architecture demonstrates the workflow of DIL protocol. According to this figure, all nodes supporting the related ledger type listen for events from the Initiator application (step 1) and upon reception of an event either create an entry in the shared state or examine the correctness of the state information (step 2). *The IL state management* assigns different roles (i.e., either active or standby) to each node in the consortium based upon a deterministic algorithm. According to these roles, the active node is supposed to continue with passing the transaction to the Responder application whereas the backup nodes are only responsible for monitoring the whole process. The active node notifies the *IL state management* about the acceptance of the role before proceeding with the operation since passing the information to Responder ledgers may take a long time especially in the multi-ledger mode (step 4). After that, the chosen node performs Responder operation (steps 5-6, for sake of simplicity, Figure 15 only represents the one-to-one mode) and write the outcome of the operation to the shared state (step 7). In case of time out in step 4 or 7, the first backup node will replace the active node and repeats the step 4-7. On the Initiator side, the validator node is selected using the same algorithm as in step 3 (step 8) and passes the transaction outcome to the Initiator application (step 9). Unlike the Responder operation, the Initiator operation is relatively fast and thus the validator node does not require writing an additional acceptance notification to the IL state management. The whole operation will be concluded by updating the shared state (steps 10-11). In a similar manner to step 4 and 7, if each of final steps (10-11) times out, the first standby node will take over the operation.

5. Initial communication platform

This chapter provides an overview of the PHOENIX Secure and Persistent Communications Layer, and describes its interactions with the Universal Secure Gateway, the Interledger Solution, and the Privacy Protection Enforcement (PPE) Smart Contract Management.

The SPC Layer adopts a data-centric approach based on federated DLTs to achieve a higher degree of persistency, traceability, availability, integrity, and interoperability in the context of data communications. The SPC Layer leverages the use of the Interledger component that facilitates the integration of multiple legers, resulting in cohesive storage platform where different types of ledgers can be used simultaneously. This component can be used for either transferring data among different ledgers, storing data hashes, or exchanging digital assets. A detailed description of the Interledger component is provided in Chapter 3.2.

Moreover, the SPC Layer deploys a TAXII server that aims to handle CTI exchange from PHOENIX components generating CTI towards the DLT of the SPC Layer. This will provide secure federated communications among LSP level PHOENIX components as well as between those components and the I2SP Platform. More information about the TAXII Server, TAXII Client Implementation are given in Chapter 2. The SPC also leverages the use of the Universal Secure Gateway that has Cyber Threat Intelligence features. Indeed, this component might be able to detect anomalies in the network communications. It can communicate them to the central component of the Phoenix system responsible for collecting and sharing Cyber Threat Intelligence (TAXII Server). An example of interactions of the SPC Layer with TAXII clients such as SAPC, IMEC and the USG are depicted in Figure 16.

The service mesh will also strengthen the security by design approach of the project. For many years, the energy related infrastructure has been more traditional, cloud native communications can be leveraged since it is beyond standard approaches. A brief background on the Cloud Native Application is provided in Chapter 3.

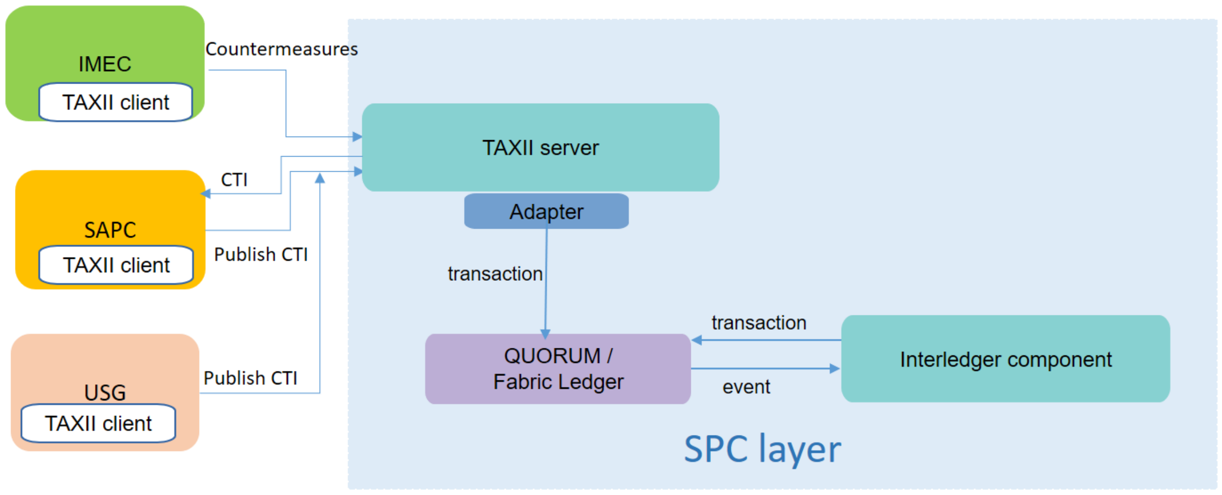

Figure 16: Example of Interactions of the SPC Layer with the TAXII clients.

5.1. Collecting data from USG

Since legacy EPES devices do not support the security and data persistency functionalities as defined in PHOENIX project, an interface device between the legacy EPES is needed. Such a device would be universal, secure and act as a gateway between the legacy EPES and the PHOENIX framework. Thus, the name: Universal Secure Gateway (USG), shown as "USG" in Figure 17.

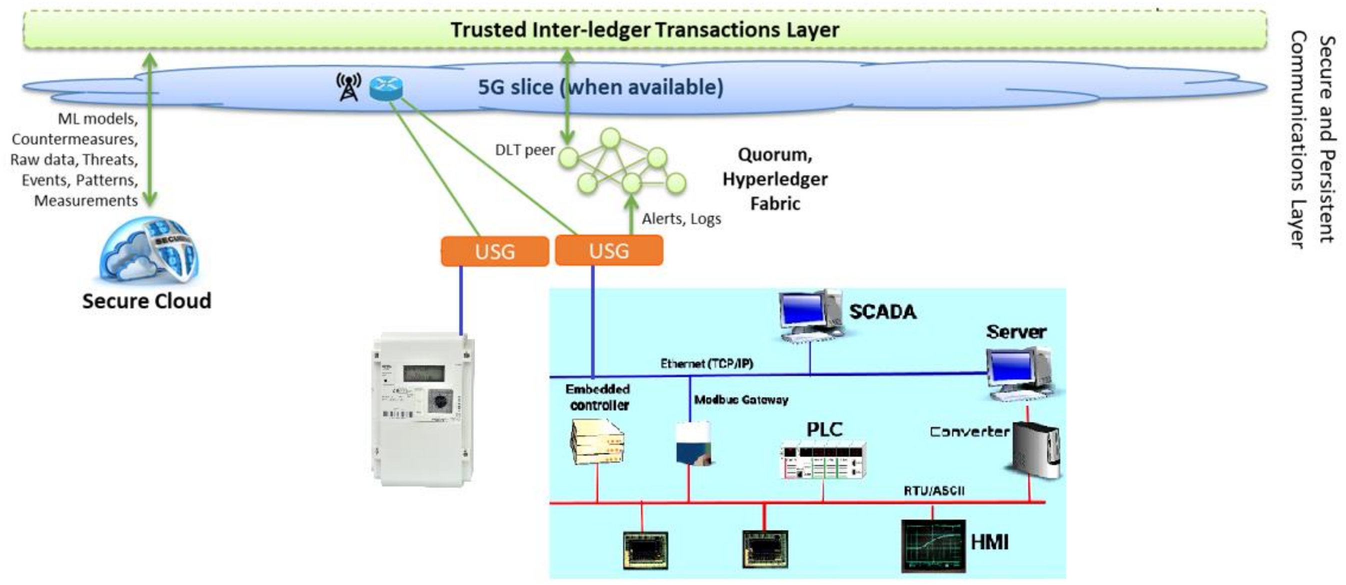
Trusted Inter-ledger Transactions Layer
5G slice (when available)
ML models, Countermeasures, Raw data, Threats, Events, Patterns, Measurements
Secure Cloud
DLT peer
Quorum, Hyperledger Fabric
Alerts, Logs
USG
USG
SCADA
Server
Ethernet (TCP/IP)
Embedded controller
Modbus Gateway
PLC
Converter
RTU/ASCII
HMI
Secure and Persistent Communications Layer

Figure 17: USG positioning in the SPC Layer.

USG hosts several CTI features, e.g., anomaly detection on the communication network. USG monitors the traffic on the various communication channels and based on the provided detection algorithms identifies the anomalies or attacks and communicates them to the central TAXII server, which is responsible for collecting and sharing CTI.

USG can serve as secure communication link between the spatially distributed legacy devices to provide secure data communication in case they do not support that feature yet.

A detailed description of the USG has been provided in D2.5: Universal Secure Gateway (Ver. 1) [47] and will be updated in D2.6: Universal Secure Gateway (Ver. 2).

5.2. Interactions with Interledger solution

In PHOENIX platform where a large amount of CTI data is generated and exchanged, DLTs are mainly used to establish agreements between different components in terms of compliance rules and governance policies for data exchange. Additionally, they significantly improve the integrity of data communication within the SPC Layer since the TAXII server stores the hash of each posted STIX object in SPC's ledger and later uses the hash to validate the correctness of STIX records maintained by the TAXII server. Therefore, DLTs enable the PHOENIX platform to achieve a higher degree of traceability, availability, integrity, and interoperability in the context of data communications.

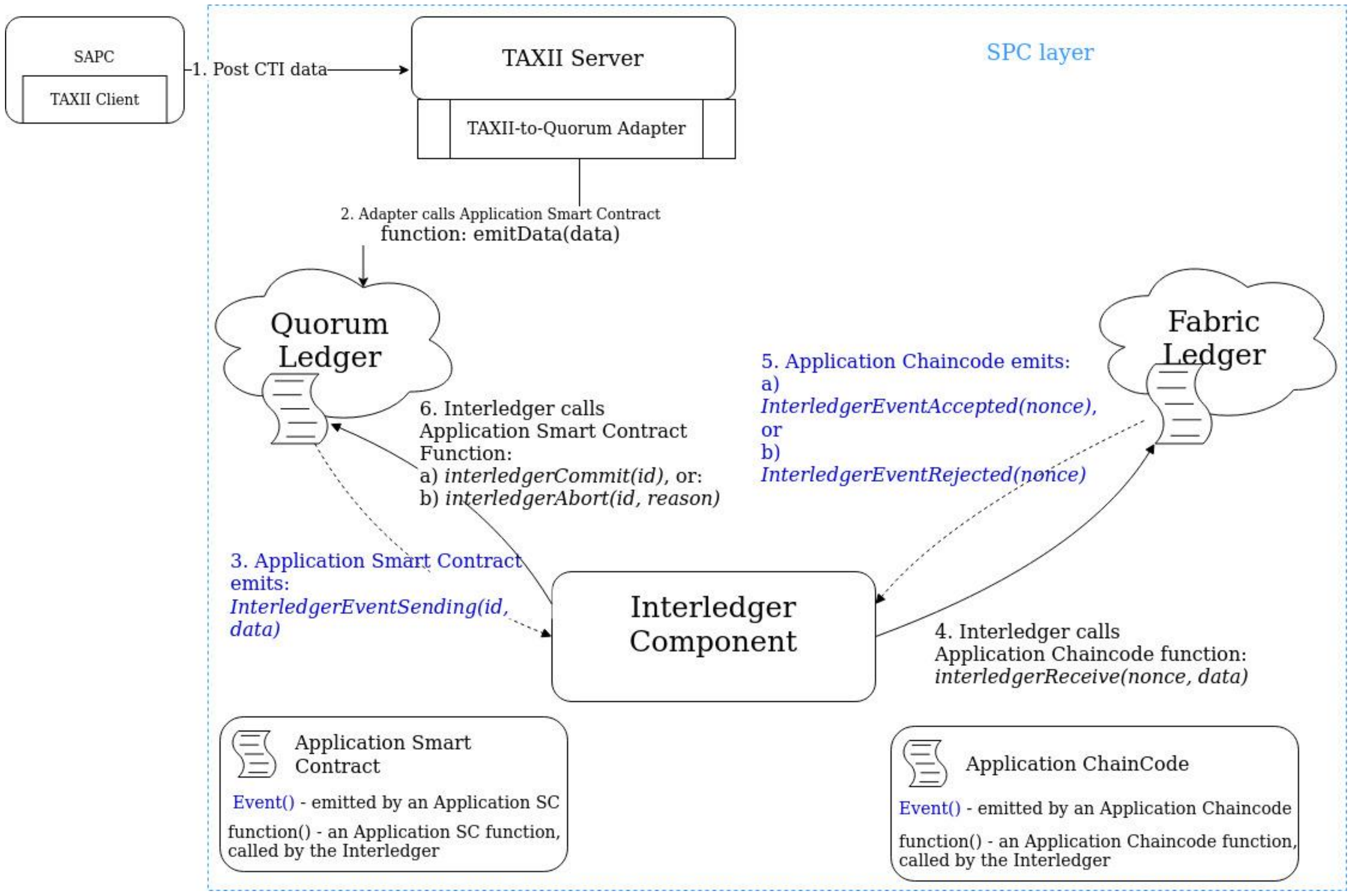
SAPC
TAXII Client
1. Post CTI data
TAXII Server
TAXII-to-Quorum Adapter
SPC layer
2. Adapter calls Application Smart Contract function: emitData(data)
Quorum Ledger
Fabric Ledger
5. Application Chaincode emits: a) InterledgerEventAccepted(nonce), or b) InterledgerEventRejected(nonce)
6. Interledger calls Application Smart Contract Function: a) interledgerCommit(id), or: b) interledgerAbort(id, reason)
3. Application Smart Contract emits: InterledgerEventSending(id, data)
Interledger Component
4. Interledger calls Application Chaincode function: interledgerReceive(nonce, data)
Application Smart Contract
Event() - emitted by an Application SC
function() - an Application SC function, called by the Interledger
Application ChainCode
Event() - emitted by an Application Chaincode
function() - an Application Chaincode function, called by the Interledger

Figure 18: An example scenario of usage of the IL component within SPC Layer.

For demonstrating the Interledger protocol, which is implemented by the IL component, we consider a scenario where SPC Layer internally utilizes two types of ledger (i.e., Quorum and Hyperledger Fabric) for persisting incoming CTI. More specifically, as described in chapter 2, the TAXII server upon reception of any STIX objects (step 1) including detected attacks, anomaly reports and proposed countermeasures, apart from persisting them in the MongoDB database, stores the hash of the object in SPC's DLTs for improving the data integrity. Figure 18 depicts an example scenario of this operation where the TAXII server is connected to Quorum ledger via TAXII-to-Quorum adapter. As shown Figure 18, the TAXII server using the adapter triggers a transaction carrying the hash of the STIX object on the Quorum ledger (step 2). Next, the Quorum node which is running the smart contract implementing the sender interface, emits an *InterledgerEventSending* event (step 3). The event carries the id parameter and data payload which is the hash of the STIX object. When the IL component catches the emitted event, extract the related information (i.e., the hash object) and passes it to the Hyperledger Fabric ledger by calling the *InterledgerReceive* function from the configured Responder application chaincode (step 4). The Responder application then sends back the transaction status on the Hyperledger Fabric ledger by emitting either of following events: 1) *InterledgerEventAccepted* that indicates the hash object is successfully written in Fabric ledger or 2) *InterledgerEventRejected* which shows that the data transfer triggered by the Quorum ledger is failed for any arbitrary reasons (e.g., network failure) (step 5). The IL component, depending on the state of transaction, then calls either *InterledgerCommit* or

InterledgerAbort function from the Initiator application SC to conclude the cross-ledger operation (step 6).

In the described scenario, the IL component can still handle a cross-ledger operation even if the roles of Quorum and Hyperledger Fabric ledger would be switched, meaning that the Hyperledger Fabric ledger acts as a primary SPC's ledger which is directly connected to the TAXII server via the TAXII-to-Fabric adapter while the Quorum network plays the role of secondary ledger.

5.3. Smart Contract Management and Interactions with PPE

PHOENIX Components will interact with the blockchain by creating and storing transactions through the usage of SCs. In GoQuorum [48], SCs are pieces of code written in Solidity language [49]. SCs can be invoked both from outside the blockchain and by other SCs depending by their visibility.

The Privacy Protection Enforcement (PPE) Component, as described in D2.1 [1], allows personal data management in compliance with the PRESS Framework [50]. It will be composed by two main subcomponents:

- **PPE-Core**: A standalone component that will be able to detect events, manage consent and permissions required by systems' participants, take track of those requests, and notify other components according to specific rules.
- **PPE-SmartContracts**: A set of smart contracts that will expose functionalities through which other SCs and PHOENIX Components will be able to access PPE features.

The deployment diagram presented in Figure 19 helps to understand how PPE component will be physically deployed on the PHOENIX system.

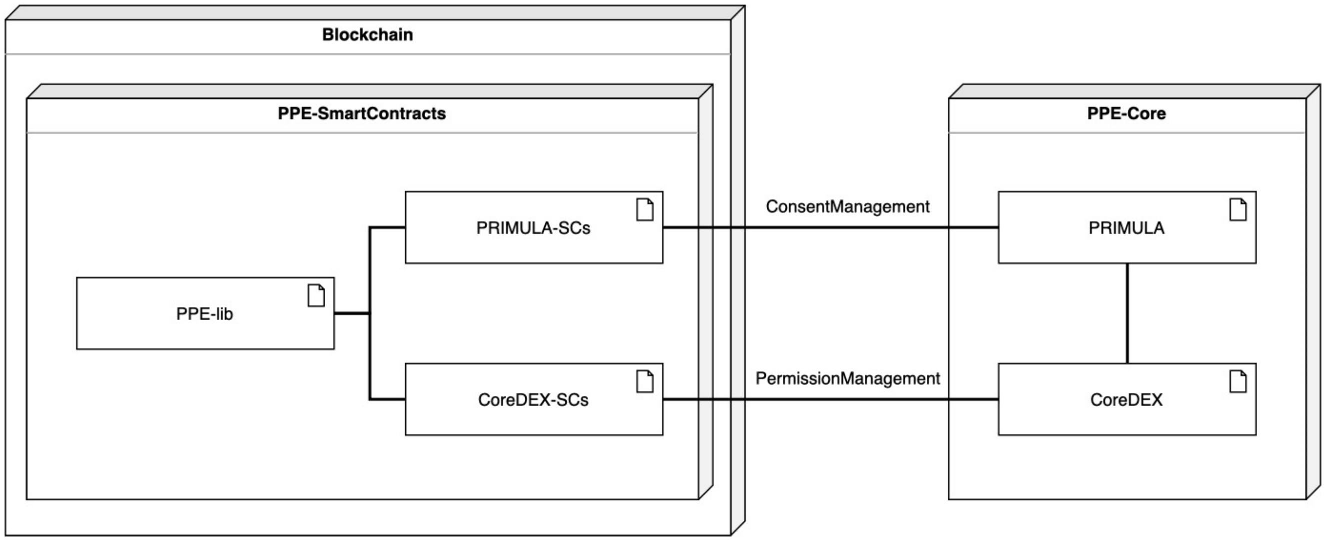

Figure 19: PPE Deployment Diagram.

PPE-SmartContracts will be deployed on the GoQuorum blockchain and the PPE component will interact with them in order to provide the following macro functionalities:

- Consent Management,
- Permission Management,
- Data Access Tracking,
- Notification.

PPE-SmartContracts development will end up with three main artefacts:

- PRIMULA-SCs: A set of smart contracts for consents management.
- CoreDEX-SCs: A set of smart contracts for permissions management.
- PPE-lib: A shared library between PRIMULA-SCs and CoreDEX-SCs.

The first two artefacts will be developed in such a way that they will be accessible also through other SCs and by the outside of the blockchain, whereas the PPE-lib will be maintained private and so not accessible from other SCs or PHOENIX Components.

PPE Smart Contracts will emit solidity events. PPE events will store arguments passed in the transaction logs. These kinds of events will be accessible using address of the contract that generates them. This kind of events will contain information like consents and permissions grant/revoke or other PPE-specific data.

For the sake of clarity, further details on data model and PPE specifications will be available on D4.3 “Cross-GDPR sensitive data exchange toolbox” (planned delivery on M22 – June 2021); however, a first draft of the class diagram with involved entities and associated methods is presented in Figure 20. As can be inferred from the diagram, there will be four main entities:

- **Data**: this object is created when a data registration request is received by PPE. It contains data metadata.
- **Consent**: represents the status of the consent assigned from the data owner to the processing of specific data.
- **Permission**: when a new data exchange request is received, PPE is responsible to agree/disagree on that data exchange.
- **Log**: every time data is accessed, this information will be logged inside the blockchain.

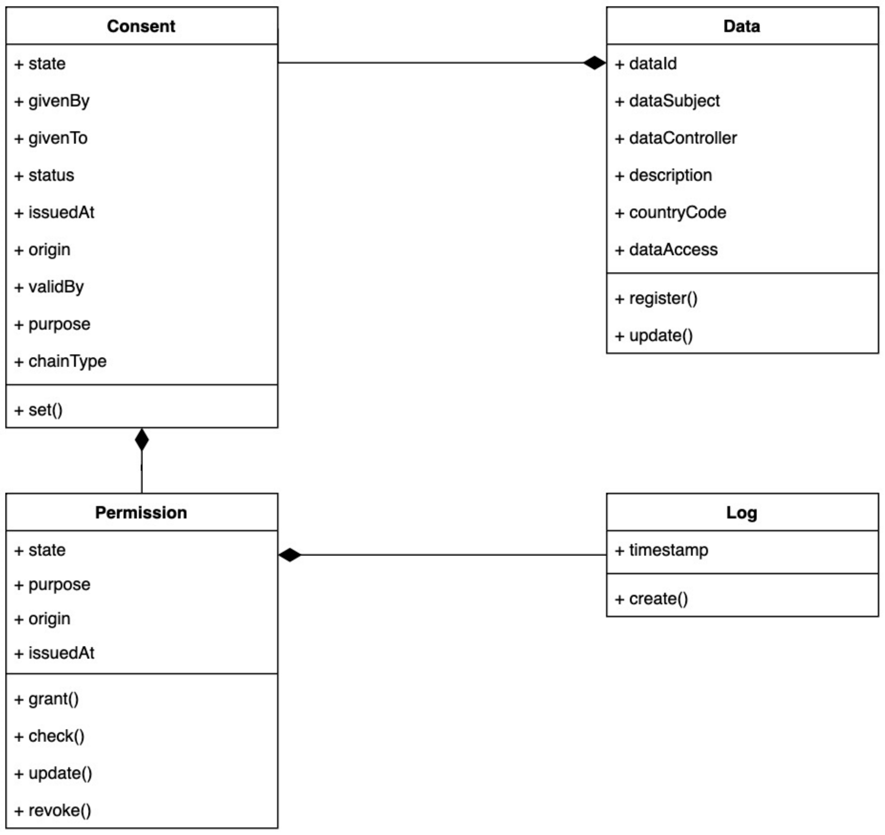

Figure 20: PPE Class Diagram.

6. Resilience Enhancement Methodology

To increase the resilience of the cyber-physical power system, we proposed the Resilience Enhancement Methodology (REM) as described in D2.2 [3]. It ensures the availability of automation functions using an implementation based on the Double Virtualization (DV) approach, opposed to the implementation in dedicated devices as in traditional automation systems. A use case related to LSP1 is defined and described in this chapter, followed by an overview of the concept of the proposed REM. Finally, a description of the design and implementation of the REM is provided.

6.1. Use case

Automation systems in electrical grids provide many functionalities required for the reliable and safe operation of the infrastructure, both under normal and under fault conditions. We chose Fault Localization, Isolation and Service Restoration (FLISR) as an example in this use case, which is a fundamental functionality to ensure reliable grid operation. Although distribution systems are often meshed, under normal operating conditions they are operated in a radial configuration, which is established by a corresponding configuration of switches. However, in case of faults in the electrical grid, the network can be reconfigured, and lost loads can be resupplied. The reconfiguration requires a localization of the fault, and the identification of a suitable reconfiguration scheme to resupply lost loads, which is composed of a sequence of switching actions. Finally, the sequence is sent to remote-controlled switches in the grid, which are switched on or off.

FLISR is one of the critical functionalities provided by the automation system under fault conditions, i.e., under conditions, where the affected power system is even more vulnerable to any additional disturbances. Attackers exploiting this condition to launch a cyber-attack on this functionality (or launching a coordinated cyber-physical attack on the infrastructure) could achieve a more severe disruption of the grid operation compared to a cyber-attack's impact under normal conditions.

The proposed REM uses the DV approach to ensure the availability of FLISR:

- In case of faults in the electrical grid and potential failures of devices in the automation system, virtualized functions can be redeployed in available devices.
- In case of event detections indicating cyberattacks, devices or DV Assets that may be corrupted can be switched off and virtualized functions can be redeployed in available devices.

6.1.1. Relation to ASM threat scenarios

The use case deployed in the context of this task is related to threat scenario ASM_SCADA_RTU, as described in D1.2: EPES threat modelling & analysis of new threats [51]. A compromise of the Supervisory Control and Data Acquisition (SCADA) system would enable an attacker to compromise or disable the FLISR functionality in this scenario. REM intends to mitigate this threat by applying DV and deploying the virtualized functions on a cluster of devices. Even if one of these is devices is then compromised, the DV approach allows for shutting down a compromised device upon detection.

6.2. Design and implementation of the Resilience Enhancement Methodology

The REM implementation includes two main parts: functions and data layer, which are specific to the use case; and DV, which offers the framework for virtualization and distribution of the relevant functions. This section describes the exemplary design and implementation for the use case described in section 6.1.

6.2.1. Functions

Fault Detection and Localization and **Service Restoration** are the relevant functions for the defined use case. The respective functionalities are explained in this section.

Fault Detection and Localization

Today's society relies on electrical energy more than ever before and expects uninterrupted supply. Given that up to 80% of all outages in distribution networks are consequence of faults or short circuits (due to equipment malfunction, weather conditions, animal contact, poor vegetation management etc.) a prompt fault localization is of paramount importance for electrical utilities. To achieve this functionality, Fault detection and Localization Algorithm (FLA) will be implemented in the context of PHOENIX project. The operation of FLA is based on the compensation theorem from circuit theory and is presented in more detail in the following subchapter. Such localization of the fault can help to expedite the repair of faulty components and significantly accelerate the power restoration process since the repair crew does no longer need to manually investigate the line or rely on reports from offline customers to localize the fault. In turn, this can help the Distribution System Operator (DSO) to minimize the number of affected customers, with isolation of the faulted area, and consequently improve the power quality indices such as System Average Interruption Duration Index (SAIDI) and System Average Interruption Frequency Index (SAIFI). Furthermore, since the FLA is also capable of detecting and localizing faults that are of temporary nature, the DSO can also investigate those in more detail. The granular measurements from Phasor Measurement Units (PMU) provide enhanced insight into the fault conditions and fault type. This is important since the temporary faults usually pass unnoticed whereas now the DSO can check the location of such faults and potentially identify malfunctioning equipment (due to poor insulation for instance) and with predictive maintenance even avoid potentially enduring faults in the future.

The goal of the method is to estimate the location of events in Distribution Networks (DN) with as little as two PMUs, utilizing compensation theorem from circuit theory. Measurements of networks' pre-event and post-event steady state synchro-phasors are used alongside known load profiles, network topology and line parameters to calculate voltages at every bus as seen from each PMU. These two sets of voltages are then compared and the bus where the difference is minimal is identified as a source of the event.

Optimal observability of the feeder using just two PMUs is achieved when one of them is installed at the beginning of the main feeder (primary substation) and the other one is placed at the end of the feeder (secondary substation). However, it is worth pointing out that in configuration with just two installed PMUs only the faults on the main feeder can be correctly identified and localized. When a fault occurs on a branch that is not a part of the main feeder, the bus, from which the lateral stems off, will be identified as faulted instead. In DNs such localization is usually sufficient, since laterals are generally short, and the exact localization does not help reduce the fault mitigation time significantly. Nevertheless, if more precise localization would be needed, one could achieve it with the introduction of PMUs at the laterals' ends. Extension of the method for the case with an arbitrary number of PMUs is presented in [36].

To summarize, the main features relevant for the implementation of FLA on the proposed framework are:

- Synchro-phasor measurements of voltages and currents acquired from as little as two PMUs are sufficient for localizing the fault on the main feeder.
- Model-driven method, meaning that topology and parameters of lines are assumed to be known.
- The rest of the information collected from the feeder, such as load consumption, can be in the form of pseudo measurements from either Power Quality Meters (PQMs) or Smart Meters (SMs). The term pseudo-measurement is used for those devices to emphasize their lower reporting rate compared to the PMU and to highlight that the measurements of PQM and SM are not as precisely time-stamped as in the case of PMU.

Service Restoration

The occurrences of natural disasters and targeted attacks on the distribution grids have caused large scale outages. Making DNs more resilient enables them to withstand and recover from these High Impact Low Probability (HILP) events and ensures continuous supply of power to the end customers. An HILP event introduces severe and rapidly changing circumstances, causing multiple outages in the network and creating large de-energized sections [52]. Furthermore, DNs need to be designed to be resilient not only to regular single faults but also against multiple faults. This can be achieved by implementing an efficient and fast Service Restoration (SR) scheme for HILP events.

A typical SR scheme for DNs, after successful fault detection and isolation, should be able to perform the following.

1) Restore as many out-of-service customers as possible in a minimum time, by providing a sequence of operation to the switches. Tele-controlled switches should be preferred in the re-powering process to reduce the restoration time.
2) Consider the priority of the loads and restore the most crucial customers (hospitals, cellular base stations, the gas network facilities, and other critical infrastructures) first.
3) Preserve radiality of the grid with every switching operation prescribed in the sequence.
4) Maintain the voltage of the grid as per the limits imposed in the grid codes of the specific country.
5) Satisfy loading constraints of the lines and substation loading.

One of the major challenges in designing a service restoration scheme to cope with the HILP events is that, in addition to the aforementioned attributes, the SR scheme should also react to rapidly changing system conditions. It should be able to consider, in real time, the uncertainties in the power generation and load demand to avoid a possible network congestion while restoring the grid. Furthermore, it should adapt to changes in the network topology as subsequent multiple faults may occur due to the propagation of the HILP events. We have chosen the rule-based SR approach proposed in [53] for the implementation in REM.

The proposed method uses graph theory to find the optimal restoration path and provides a sequential restoration scheme prioritizing the re-powering of critical loads. It also takes into account the volatility of the power generation and consumption, to better understand the current loading conditions of the grid while defining the SR sequence. This is achieved by utilizing a state estimation algorithm to quantify the congestion of the grid and power losses, leveraging on the incorporation of real time measurements coming from the field devices or forecasts. Enabling the SR to react to latest grid loading conditions and sudden events contributes to the real time applicability.

6.2.1. Double Virtualization

DV has been proposed in the H2020 SUCCESS (https://success-energy.eu/) project to provide resilience by design to monitoring and control systems to mitigate both failures and targeted attacks [54] and has been refined in the context of subsequent research projects [55]. The DV approach leverages on the availability of Intelligent Electronic Devices (IED), which are used to monitor and control EPES infrastructures. In traditional systems, functions are implemented in dedicated devices, and the respective functionality is lost if the corresponding device fails due to attacks or natural disasters. Following the DV approach, function and data layer are virtualized and no longer dedicated to a specific device. Instead, functions can be re-allocated between all available devices. An example is provided below, considering the functionality of State Estimation (SE). SE calculates the estimated system state based on all available PMU measurements and their uncertainties. For this, synchrophasors from PMUs are sent to Phasor Data Concentrators (PDC) and then forwarded to the control centre where the SE is performed (see Figure 21).

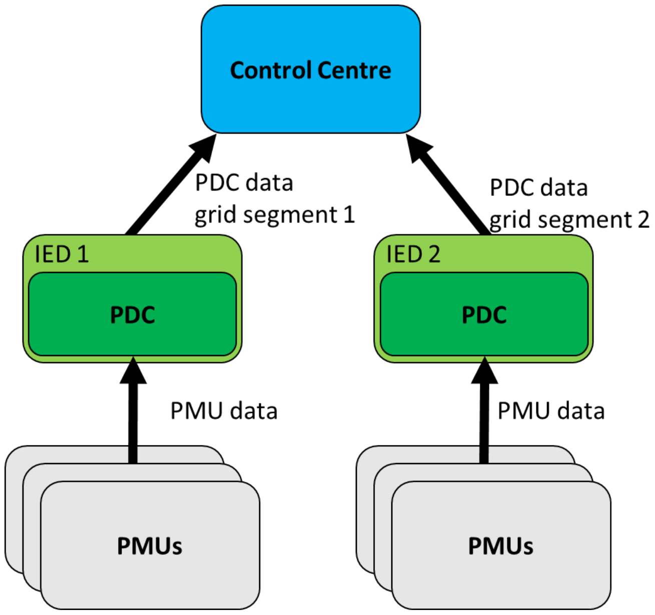

Figure 21: DV use case example - grid monitoring based on PMU measurements.

In case of a PDC failure, all measurements from the corresponding PMUs would be lost, as shown in Figure 22, and the quality of the SE results would decrease accordingly.

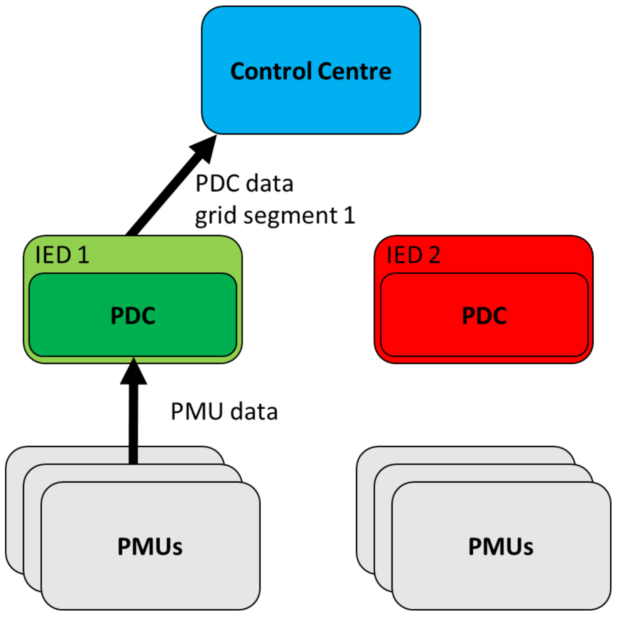

Figure 22: DV use case example - PDC failure without DV.

Following the DV approach, the PDC's function (collecting phasors from PMUs and forwarding them to the control centre) is virtualized and can be re-allocated between each of the available devices. In case of a PDC failure, the corresponding PMU measurements are collected by a different PDC and the availability of synchrophasors for the SE is ensured (see Figure 23)

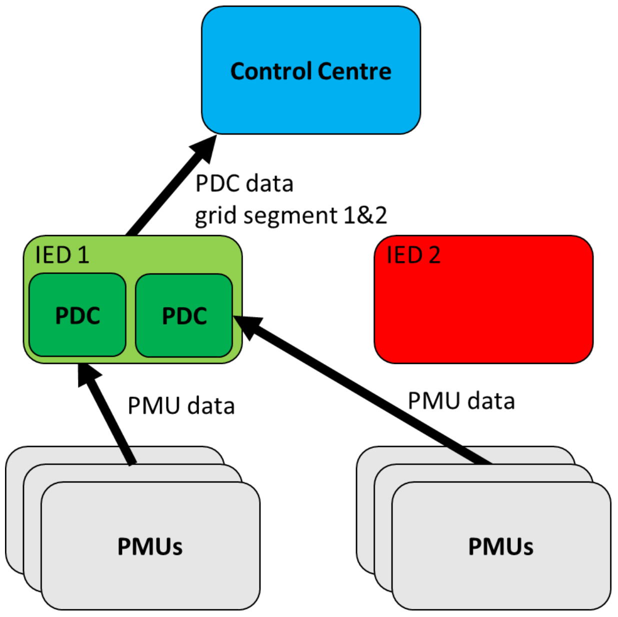

Figure 23: DV use case example - PDC failure with DV.

The aforementioned example (and other potential applications of DV) requires a meshed network of intelligent devices as the PDCs that are collectively responsible for operation of the grid. Without redundant devices for hosting the automation functions, the benefits of DV cannot be realised. From the example, the following prerequisites are identified for potential DV applications:

- DV can only be applied to increase the resilience of EPES automation systems whose operational intelligence is hosted by multiple devices.
- The control and monitoring algorithms (automation functions) should be virtualized and deployed as either executables or automated scripts so that they can be migrated or re-instantiated in different devices.

Design approach to Double Virtualization

DV can be conceptualized as an approach to the Observe-Orient-Decide-Act (OODA) pattern, first introduced by John Boyd [56]. By using this pattern, the intent is to enable DV entities to react preventively to potentially harmful situations by anticipating them through observation.

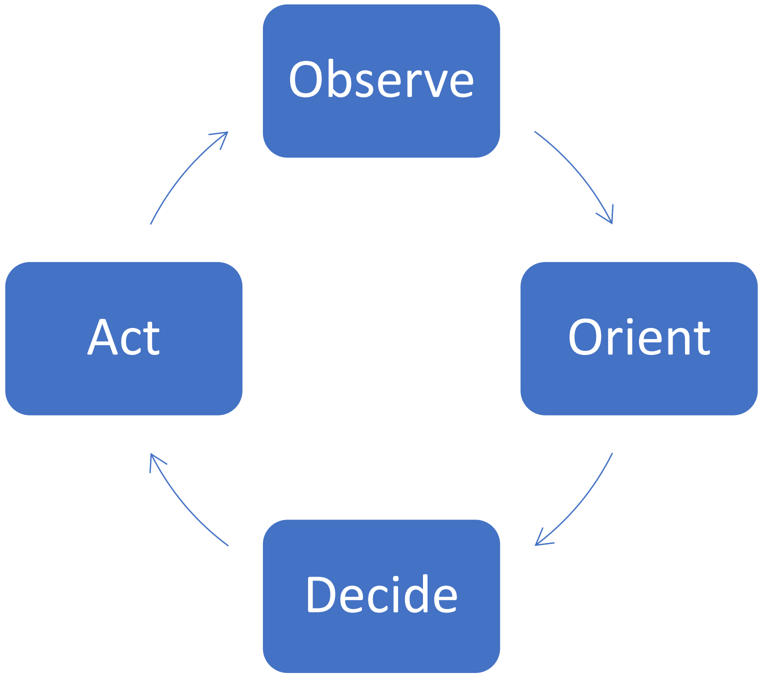

Figure 24: Simplified OODA Diagram

The steps of the OODA loop are mapped to DV tasks as follows:

- **Observe**: In this step, internal or external sensors collect information about the state of DV entities.
- **Orient**: DV incorporates interpreters for the collected data to contextualize it.
- **Decide**: Based on the collected and interpreted data, specific algorithms identify suitable actions (e.g., migration or re-instantiation of virtualized functions).
- **Act**: The mitigation actions identified in the previous step are applied.

Implementation of Double Virtualization

SUCCESS relied on Calvin[1] for the implementation of DV. However, Calvin did not maintain an active community and has been discontinued. For that reason, Node-RED[2] will be used instead of Calvin for the DV implementation in PHOENIX.

Node-RED is an Integrated Development Environment (IDE) and execution engine focusing on IoT applications. It provides a runtime based on Node.js, where programs are implemented as sets of nodes, which are wired together and form flows. Node-RED has an active community and offers many customized nodes e.g., to provide communication interfaces, integration of databases and execution of external scripts. Finally, Node-RED can be used across several operating systems and architectures, turning it into an excellent tool to be used in heterogeneous systems.

Node-RED facilitates the virtualization of functions, either by graphically and easily creating flows that perform the required logic (embedded on the Node-RED application) or, by enabling the execution of several types of scripts (Bash, Python, etc.) while keeping them monitored and controlled.

For the initial REM implementation, we use the DV implementation presented in [55] as a starting point. The aforementioned DV implementation is based on two types of components:

- **DV Administration and Management** (DVA&M): The DVA&M is responsible for the coordination and management of all available DV Assets and the deployment virtualized functions on them. DVA&M is a centralized component, deployed on a dedicated device.
- **DV Asset**: Each DV Asset corresponds to a physical device, which is running a Node-RED runtime, enabling the DVA&M to deploy virtualized functions as Node-RED flows.

While this implementation utilizes the redundancy of the set of DV Assets to eliminate the single point of failure of deploying functions on dedicated devices, it creates another single point of failure in form of the DVA&M. Further refinements are planned for the final implementation. One of these refinements will include distributing the DVA&M functionality to eliminate the single point of failure.

6.2.2. Implementation of the Resilience Enhancement Methodology

For the implementation of REM, the DV framework is provided by one DVA&M and at least one DV Asset. The DVA&M is responsible for administration and monitoring of the DV Assets and is configured according to the available DV Assets.

For the DV Assets, different flows are defined as described in Table 2. The “Interface and storage” flow is deployed on each DV Asset, receives data from the DVA&M and stores it in a local database. All other flows are normally running on one of the available DV Assets. The respective flow calls and monitors a Python script, which runs continuously. Additionally, the flow receives the output of the scripts, which

[1] https://github.com/EricssonResearch/calvin-base
[2] https://nodered.org/

are the location of the fault or the switching sequence for network reconfiguration here. These outputs are then forwarded to the DVA&M.

Table 2: DV Asset - overview of relevant flows

Flow	Description	Deployment
Interface and storage	Receives data from DVA&M stores it in the local database	On each DV Asset
Fault localization	Calls and monitors a local Python script for fault localization, receives results and forwards them to DVA&M	On one DV Asset
Service Restoration	Calls and monitors a local Python script for Service Restoration, receives results and forwards them to DVA&M	On one DV Asset

6.3. Outlook

As explained earlier, a shortcoming of the current DV implementation is the DVA&M, which is a centralized component, deployed on a dedicated device and thus acts as a single point of failure. This shortcoming can be overcome by distributing the DVA&M functionality in the DV Assets with the help of a consensus mechanism to enable coordination between all DV Assets.

An update on the proposed REM and the implementation will be provided in D2.4, including the distributed DVA&M.

7. Simulation Studies

This chapter introduces the laboratory set-up used for the implementation and evaluation of REM and presents our approach to the evaluation. Furthermore, we give an outlook on planned improvements and refinements of the implementation.

7.1. Laboratory set-up

The laboratory set-up comprises two main parts:

1. A model of the electrical grid, for which the REM approach is utilized; and
2. the REM infrastructure, including DV and relevant monitoring and control functions.

7.1.1. Grid model

The considered distribution network is 23.9 kV, 3-phase, ungrounded, meshed network located in Italy and operated by ASM Terni, which is part of LSP1. The network is composed of both underground cables and overhead lines, and includes loads and RES, specifically photovoltaic units.

The proposed FLISR methodology will be tested in the simulation environment using the Real-Time Digital Simulator (RTDS), which currently offers the closest approximations to the conditions in real-life grids. The meshed network is effectively divided into four independent radial sub-networks with the help of breakers.

Due to computational limitations of the simulator some simplifications of the network are required. Therefore, the loads are aggregated with the consumption equal to sum of all corresponding loads in the real grid. Loads are ungrounded and connected in symmetrical Δ-connection with assumption of constant impedance. The lines are modelled with equivalent π-sections, which is the common approach for distribution networks. The upstream of the grid is presented with a stiff power-sources, whereas the Renewable Energy Sources (RES) are modelled as constant current sources.

7.1.2. REM infrastructure

The REM infrastructure comprises the software components described in section 6.2 and the hardware, on which these components are deployed. We are using single board computers for the deployment of the software components, as described in Table 3.

Table 3: Specification of single board computers for REM deployment.

Model	Raspberry Pi Model 4B
CPU	Quad-core Cortex-A72 (ARM v8)
RAM	8GB LPDDR4-3200 SDRAM
Storage	32GB SD card

In the current set-up, three single board computers are used, one as DVA&M, and two as DV Assets. The DVA&M also acts as an interface between the RTDS simulation and the DV Assets, as shown in Figure 25. The blue boxes represent the single board computers, where only one of the DV assets is shown here. Each of them runs a Node-RED runtime containing several flows. One set of flows, both in the DVA&M and DV Asset, is responsible for the administration and management of the DV infrastructure. This includes registration and monitoring of the DV Assets and migration of functions between DV Assets. The dashed arrows between these flows represent the communication required for administration and management. The solid arrows represent process data flows, in this case simulation data from RTDS and control commands to RTDS. A dedicated flow is set up in the DVA&M for forwarding the data to all DV Assets, where it is received and stored in a database. Each function is implemented as a separate flow and a corresponding Python script, where the flow starts and controls the Python script (indicated by dashed arrow). The script directly receives data from the database, results are written in the database and/or sent to the corresponding Node-RED flow. Finally, the results are sent to RTDS via the flow in the DVA&M.

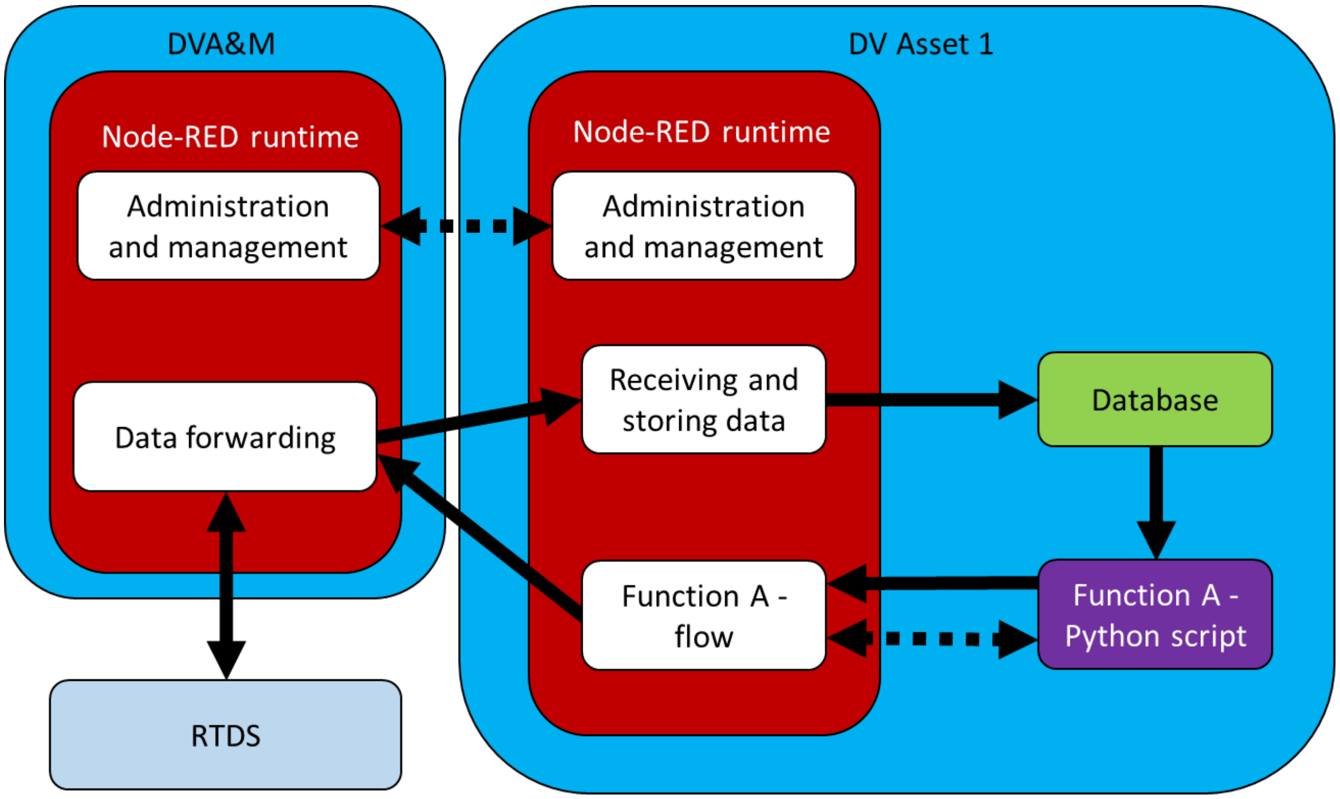
DVA&M
Node-RED runtime
Administration and management
Data forwarding
RTDS
DV Asset 1
Node-RED runtime
Administration and management
Receiving and storing data
Database
Function A - flow
Function A - Python script

Figure 25: Simplified REM infrastructure and laboratory set-up.

7.2. Evaluation

This section presents the approach to the evaluation of the proposed REM implementation. The final results based on the proposed evaluation criteria will be included in D2.4.

7.2.1. Scenario

As an initial evaluation scenario, we assume a fault occurring in one of the feeders. The fault is happening close to the beginning of the feeder; thus, all downstream loads are disconnected. Additionally, we assume one of the DV Assets is located and supplied by the feeder affected by the fault. As soon as the fault occurs, the DV Asset fails since it is no longer powered.

7.2.2. Criteria

To evaluate the performance of the proposed REM implementation, we will use the criteria given in Table 4.

Table 4: Criteria for evaluation on the REM implementation

Criterion	Description
Functionality maintained	The FLA and SR functionality are maintained; the location of the fault is identified correctly, and the reconnectable loads and DV Assets are resupplied
Time to restore functionality	The time difference between failure of a DV Asset and redeployment of its active flows in a different DV Asset

In addition to these criteria, we provide a qualitative evaluation of the implementation based on the requirements defined in D2.2 [3], discussing current shortcomings and potential improvements.

8. Conclusion

This deliverable provided the update on the Secure and Persistent Communications Layer in PHOENIX. An overview of the initial communication platform was given, including the SPC Layer description and its interaction with other key components as the USG, the Interledger solution and the PPE. To increase the availability and reliability of the PHOENIX core components and foster communications resilience, the platform deployment of PHOENIX follows the cloud native paradigm with the adoption of container-based operation and relevant container orchestration processes.

PHOENIX relies on TAXII and STIX to exchange CTI information, with a TAXII server deployed at LSP level as part of the SPC Layer. The exchange of CTI is realized by a publish/subscribe mechanism between the TAXII server, and the clients implemented in the other components of the PHOENIX platform. The USG is one of these components, deployed at LSP level to provide cyber-attack detection capability (among other functionalities) with STIX-formatted CTI as output.

For the persistent and immutable storage of this CTI, DLTs are utilized. The proposed DLT-based Interledger solution improves traceability, availability, integrity, and interoperability of the data exchange. Communication between various ledgers can be realized with the current implementation of the Interledger solution while additional ledgers can be integrated. This enables the CTI exchange between different EPES infrastructures while maintaining inclusiveness to different ledgers.

Additionally, we have investigated “by-design” measures to increase EPES resilience and proposed the Resilience Enhancement Methodology for that purpose, by exploiting existing redundancies (in the physical infrastructure) and creating additional redundancy via virtualization (in the automation and control system). The current implementation, a relevant use case and an outlook to further improvements of the implementation have been provided. The approach to the evaluation and the corresponding laboratory set-up have been described.

The presented solutions are work in progress and the final results will be given in the last iteration of the SPC Layer deliverables, which is D2.4: Secure and Persistent Communications Layer (Ver. 2).

9. References

[1] H2020 PHOENIX, "D2.1: PHOENIX platform architecture specification," 2020.

[2] H2020 PHOENIX, "D2.2: Secure and Persistent Communications Layer (Ver. 0)," 2020.

[3] H2020 PHOENIX, "D6.1: PHOENIX Integration guidelines and integrated platform (Ver. 0)," 2020.

[4] OASIS-Open, "TAXII™ Version 2.1 - Section 6 TAXII™ API - Channels," 2020. [Online]. Available: https://docs.oasis-open.org/cti/taxii/v2.1/cs01/taxii-v2.1-cs01.html#_Toc31107545. [Accessed 28 Jun. 2021].

[5] OASIS-Open, "TAXII™ Version 2.1," 2020. [Online]. Available: https://docs.oasis-open.org/cti/taxii/v2.1/cs01/taxii-v2.1-cs01.html#_Toc31107501. [Accessed 28 Jun. 2021].

[6] OASIS, "OASIS TC Open Repository - cti-taxii-server/medallion/," GitHub, 2020. [Online]. Available: https://github.com/oasis-open/cti-taxii-server/tree/master/medallion. [Accessed 28 Jun. 2021].

[7] Open Source Initiative, "The 3-Clause BSD License," [Online]. Available: https://opensource.org/licenses/BSD-3-Clause. [Accessed 29 Apr. 2021].

[8] VMware, "RabbitMQ," VMware, 2007-2021. [Online]. Available: https://www.rabbitmq.com/. [Accessed 29 Apr. 2021].

[9] Apache, "Apache Kafka," [Online]. Available: https://kafka.apache.org/. [Accessed 29 Apr. 2021].

[10] MongoDB, "MongoDB NoSQL database," 2021. [Online]. Available: http://www.mongodb.org. [Accessed 26 Apr. 2021].

[11] Blockgeeks, "What Is Quorum Blockchain? A Platform for The Enterprise," 2020. [Online]. Available: https://blockgeeks.com/guides/quorum-a-blockchain-platform-for-the-enterprise/. [Accessed 28 Jun. 2021].

[12] P. Thummavet, "Demystifying Hyperledger Fabric (1/3): Fabric Architecture," 2 May 2019. [Online]. Available: https://medium.com/coinmonks/demystifying-hyperledger-fabric-1-3-fabric-architecture-a2fdb587f6cb. [Accessed 28 Jun. 2021].

[13] J. Reschke, "The 'Basic' HTTP Authentication Scheme," *Work in Progress, draft-ietf-httpauth-basicauth-update-07,* 2015.

[14] K. Nane and P. C. Quint, "Understanding Cloud-native applications after 10 years of cloud computing - A systematic mapping study," *Journal of Systems and Software,* pp. 1-16, 2017.

[15] N. Herbst, S. Kounev and R. Reussner, "Elasticity in cloud computing: what it it and what it is not," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), USENIX*, San Jose, CA, 2013.

[16] A. Bondi, " Characteristics of scalability and their impact on performance," in *Proceedings of the 2nd International Workshop on Software and Performance*, New York, NY, USA, 2000.

[17] M. Fowler, "Microservices - a definition of this new architectural term," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed 28 Jun. 2021].

[18] C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li and S. Dustdar, "A methodology for architecture and deployment of cloud application topologies," in *IEEE International Symposium on Service Oriented System Engineering (SOSE),* , 2014.

[19] C. Fehling, F. Leymann, R. Retter, W. Schupeck and P. Arbitter, Cloud Computing Patterns, Springer, 2014.

[20] K. Indrasiri, “Service Mesh for Microservices.,” 2017. [Online]. Available: https://medium.com/microservices-in-practice/service-mesh-for-microservices-2953109a3c9a. [Accessed 28 Jun. 2021].

[21] NIST, “Building Secure Microservices-based Applications Using Service-Mesh Architecture,” https://doi.org/10.6028/NIST.SP.800-204A, May 2020.

[22] W. Morgan, “What’s a service mesh? And why do I need one?,” 2017. [Online]. Available: https://buoyant.io/2020/10/12/what-is-a-service-mesh/. [Accessed 28 Jun. 2021].

[23] I. Tsoumas, C. Symvoulidis, D. Kyriazis and P. Gouvas, “Modelling 5G Cloud-Native Applications by Exploiting the Service Mesh Paradigm,” in *EMECS* , 2020.

[24] H2020 MATILDA 5G, “Project website,” 2019. [Online]. Available: https://www.matilda-5g.eu/. [Accessed 28 Jun. 2021].

[25] Kubernetes, “What is Kubernetes?,” [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 21 Apr. 2021].

[26] E. Brewer, “Kubernetes and the path to cloud native,” 2015. [Online]. Available: https://dl.acm.org/citation.cfm?id=2809955. [Accessed 21 Apr. 2021].

[27] L. . Baumann, S. . Benz, L. . Militano and T. M. Bohnert, “Monitoring Resilience in a Rook-managed Containerized Cloud Storage System,” , 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8801967. [Accessed 21 4 2021].

[28] L. Mercl and J. Pavlik, “Public Cloud Kubernetes Storage Performance Analysis.,” in *International Conference on Computational Collective Intelligence*, 2019.

[29] Ceph Foundation, “CEPH Storage,” 2021. [Online]. Available: https://ceph.io. [Accessed 21 Apr. 2021].

[30] R. Maull, P. Godsiff, C. Mulligan, A. Brown and B. Kewell, “Distributed ledger technology: Applications and implications.,” *Strategic Change,* 2017.

[31] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 2020. [Online]. Available: https://bitcoin.org/bitcoin.pdf. [Accessed 28 Jun. 2021].

[32] S. De Angelis, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri and V. Sassone, “Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain,” in *Italian Conference on Cybersecurity*, 2018.

[33] V. A. Siris, P. Nikander, S. Voulgaris, N. Fotiou, D. Lagutin and G. C. Polyzos, “Interledger approaches,” *IEEE Access,* vol. 7, pp. 89948-89966, 2019.

[34] M. Herlihy, “atomic cross-chain swaps,” in *Proceedings of the 2018 ACM symposium on principles of distributed computing*, 2018.

[35] C. Burchert, C. Decker and R. Wattenhofer, “Scalable funding of bitcoin micropayment channel networks,” *Royal Society open science,* 2018.

[36] A. Hope-Bailie and S. Thomas, “Interledger: Creating a standard for payments,” in *Proceedings of the 25th International Conference Companion on World Wide Web*, 2016.

[37] A. Culwick and D. Metcalf, “The blocknet design specification,” 2018.

[38] A. Singh, K. Click, R. M. Parizi, Q. Zhang, A. Dehghantanha and K.-K. R. Choo, “Sidechain technologies in blockchain networks: An examination and state-of-the-art review,” *Journal of Network and Computer Applications,* vol. 149, 2020.

[39] S. M. English, F. Orlandi and S. Auer, “Disintermediation of inter-blockchain transactions,” *arXiv preprint arXiv:1609.02598,* 2016.

[40] Bitcon Wiki, “Hash Time Locked Contracts,” Bitcon Wiki, 2019. [Online]. Available: https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts. [Accessed 28 Jun. 2021].

[41] R. Belchior, A. Vasconcelos, S. Guerreiro and M. Correia, “A survey on blockchain interoperability: Past, present, and future trends,” *arXiv preprint arXiv:2005.14282,* 20202.

[42] H2020 SOFIE, “Interledger Fabric Adapter,” 2020. [Online]. Available: https://github.com/SOFIE-project/Interledger/blob/master/doc/adapter-fabric.md. [Accessed 28 Jun. 2021].

[43] H2020 SOFIE, “Interledger Indy Adapter,” 2020. [Online]. Available: https://github.com/SOFIE-project/Interledger/blob/master/doc/adapter-indy.md. [Accessed 28 Jun. 2021].

[44] H2020 SOFIE, “Interledger KSI Adapter,” 2020. [Online]. Available: https://github.com/SOFIE-project/Interledger/blob/master/doc/adapter-ksi.md. [Accessed 28 Jun. 2021].

[45] H2020 SOFIE, “Interledger Ethereum Adapter,” 2020. [Online]. Available: https://github.com/SOFIE-project/Interledger/blob/master/doc/adapter-eth.md. [Accessed 28 Jun. 2021].

[46] H2020 SOFIE, “D2.7 Federation Framework, final version,” 2020.

[47] H2020 PHOENIX, “D2.5: Universal Secure Gateway (Ver. 1),” 2021.

[48] ConsenSys, “GoQuorum,” 2021. [Online]. Available: https://consensys.net/quorum/. [Accessed 28 Jun. 2021].

[49] Ethereum Revision, “Solidity Language,” 2016-2021. [Online]. Available: https://docs.soliditylang.org/en/v0.8.4/. [Accessed 28 Jun. 2021].

[50] H2020 PHOENIX, “D4.1: PRESS Framework Analysis,” 2020.

[51] H2020 PHOENIX, “D1.2: EPES threat modelling & analysis of new threats,” 2020.

[52] M. Panteli and P. Mancarella, “The grid: Stronger, bigger, smarter?: Presenting a conceptual framework of power system resilience,” *IEEE Power and Energy Magazine, Volume: 13, Issue: 3,* p. 58–66, May-June 2015.

[53] A. Dognini, A. Sadu, A. Angioni, F. Ponci and A. Monti, “Service Restoration Algorithm for Distribution Grids under High Impact Low Probability Events,” in *2020 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, The Hague, Netherlands, 2020.

[54] H2020 SUCCESS, “D2.5: The Resilience by Design concept, V2,” 2018.

[55] G. Di Orio, G. Brito, P. Malo, A. Sadu, N. Wirtz and A. Monti, “A Cyber-Physical Approach to Resilience and Robustness by Design,” *International Journal of Advanced Computer Science and Applications (IJACSA),* 2020.

[56] J. R. Boyd, *Destruction and Creation,* U.S. Army Command and General Staff College, 1976.