July 2021

D 5.1 - Interfaces and infrastructure specification of core blockchain services

The project Flexible Energy Production, Demand and Storage-based Virtual Power Plants for Electricity Markets and Resilient DSO Operation (FEVER) receives funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 864537

Project name	FEVER
ID & Title	D5.1 - Interfaces and infrastructure specification of core blockchain services
Work package:	WP5 – Toolbox for P2P Flexibility markets
Contractual delivery date:	31.07.2021
Actual delivery date:	29.07.2021
Main responsible:	Laurynas Šikšnys, FlexShape Aps
Security:	P = Public
Nature:	R
Version:	v1.0
Total number of pages:	93

Authors and reviewers

Main responsible: Laurynas Šikšnys, FlexShape Aps, laurynas@flexshape.dk

Author(s)/contributor(s): Jonas Brusokas, AAU
Torben Bach Pedersen, FlexShape
Viktor Alkalais, Dimitrios Karagkounis, Ilias Lamprinos ICOM
Gregor Černe, INEA
Christos Patsonakis, Konstantinos Votis, CERTH/ITI

Reviewer(s): Rita Dornmair (BAUM)
Christopher Schneider (SWH)

Abstract

This document presents a blockchain-based software platform for peer-to-peer (P2P) flexibility trading (P2P-FTP). P2P-FTP targets so-called energy communities (ECs) and offers a novel concept of a pseudo-currency (FlexCoins) and an auction-based P2P marketplace solution, which allows electricity consumers and/or producers offering and requesting advanced energy and/or flexibility products to/from the members (peers) of an EC. The document starts by motivating and explaining the P2P trading process using simple and intuitive so-called user stories. Then, based on these user stories, a number of P2P-FTP requirements are specified. To address these, the specifications of (1) P2P-FTP architecture, (2) distributed blockchain P2P-FTP applications for identity management, pseudo-currency, and P2P trading, (3) API for interfacing with external (FEVER) systems, (4) deployment environment, and (5) pipeline for continuous integration/deployment (CI/CD) are provided. These will serve as a basis for the subsequent practical development efforts in WP5 (D5.2-D5.3).

Keyword list

P2P, trading, flexibility trading, FlexOffers, pseudo-currency, blockchain, DLT, Energy Community.

Executive summary

This document is the FEVER deliverable D5.1, titled “Interfaces and infrastructure specification of core blockchain services”. By summarizing efforts of Tasks 5.1- 5.3, the document presents a so-called peer-to-peer (P2P) flexibility trading platform (P2P-FTP), which is an end-to-end blockchain-based solution, offered as a part of Peer-to-Peer Flexibility Trading Toolbox (WP5). P2P-FTP targets so-called Energy Communities (ECs) where active electricity consumers and/or producers (prosumers) have a need to transact energy, monetary, and (other) communal assets in the peer-to-peer (P2P) fashion without a fully trusted marketplace operator. For such communities, P2P-FTP offers a novel concept of a pseudo-currency (FlexCoins) and an auction-based P2P marketplace solution, which allows electricity consumers and/or producers offering and requesting advanced energy and flexibility products to/from the members (peers) of an EC. The document starts by motivating and explaining P2P trading process from the end-user perspective by providing simple and intuitive so-called user stories. Based on these user stories, the document draws a set of requirements for the overall P2P-FTP as well as its internal parts. Then, it presents the high-level architecture of P2P-FTP and gives specifications of 3 distributed blockchain applications (DAPPs) of P2P-FTP, which deliver the functionalities for identity management, pseudo-currency, as well as P2P trading. The document concludes by presenting an API for integrating P2P-FTP with external FEVER systems (e.g. with the flexibility trading platform of WP2 and describes a cloud-based deployment environment and a continuous integration/deployment (CI/CD) pipeline for simplified deployments of P2P-FTP components. The specifications presented in this document will guide subsequently practical P2P-FTP development efforts in WP5 (D5.2-D5.3).

Contents

1 Introduction

WP5 in the FEVER project develops a so-called *Peer-to-Peer Flexibility Trading Toolbox (P2P-Toolbox)* which can be used to enable peer-to-peer (P2P) flexibility trading in so-called *Energy Communities* (ECes) based on Distributed Ledger Technologies (DLT). While developing this toolbox, WP5 targets the following main objectives (from DoA):

- Handle privacy, anonymization and identity management issues for embedding and sharing data on blockchains
- Provide a blockchain with ad-hoc consensus, pseudocurrency, classes and libraries for trading functionalities
- Provide a blockchain network for peer-to-peer energy flexibility trading
- Provide APIs for developers to instantiate user business logic and trading contracts
- Provide Cloud services and a data management platform for delivering core blockchain services

As such, the FEVER WP5 P2P-Toolbox is defined by the following two main items provided by WP5:

- **Peer-to-Peer Flexibility Trading Platform (P2P-FTP)** i.e. a collection of generic software components, which can be customized, adapted, and physically deployed on an existing infrastructure to establish real-world P2P flexibility trading in (a) selected EC(-s).

- **P2P-FTP Methodology** which includes a background as well as a set of suggestions and recommendations on how P2P-FTP should be used in a particular context, covering essential *privacy*, *anonymization*, *data management*, and *community business* aspects.

This document focuses primarily on P2P-FTP, while covering specific methodology aspects as a part of P2P-FTP specification (in their relevant sections). As such, P2P-FTP is an independent FEVER sub-system (group of components), which has the overall purpose of providing a specialized energy flexibility trading solution for the scenarios where all market participants are treated as equal peers and no *trusted* marketplace operator exists or can be assumed. Such scenarios are envisioned primarily in emerging so-called *Energy Communities,* e.g., *Renewable Energy Communities* and *Citizen Energy Communities as defined by the EU directives* [1] and 32018L2001 [2]. We use the term *peer-to-peer (P2P) trading* to denote trading in such scenarios.

The requirements for P2P-FTP primarily stem from earlier documents (FEVER DoA, HLUC 15 - D1.1, and PUC32 - D1.2) as well as subsequent discussions/workshops, during which further system specification was performed. Based on that, P2P-FTP is defined as a collection of so-called DAPPs - decentralized applications, which run on a distributed blockchain network (as opposed to a central server). Specifically, P2P-FTP consists of the following 3 DAPPs based on the Hyperledger Fabric:

1. **Community Management DAPP** – this DAPP allows configuring EC values/assets and their members: (de-)registering the types of assets that have *communal value* to the EC; adding, modifying, revoking users and user access rights, etc. This DAPP establishes a root of trust and provides user authentication and authorization services for other DAPPs.
2. **FlexCoin DAPP** – this DAPP manages so-called FlexCoins – units of a specialized pseudo-currency used to quantify and track EC member contributions to the EC welfare. It allows tracking FlexCoin balances as well as FlexCoin transactions and exchanging FlexCoins with other assets/values within the community (e.g., fiat currencies, m^2 of houses painted, etc.).
3. **FlexTrading DAPP** – this DAPP offers a marketplace where its participants (peers) are able to trade flexibility/energy products without an intermediary. It allows configuring trading parameters (contracts), monitoring and tracking statuses, histories, and states of several parallel P2P markets and individual peers, offering some options for manual interventions/ bidding (e.g., submitting or revoking bids manually from GUI, etc.).

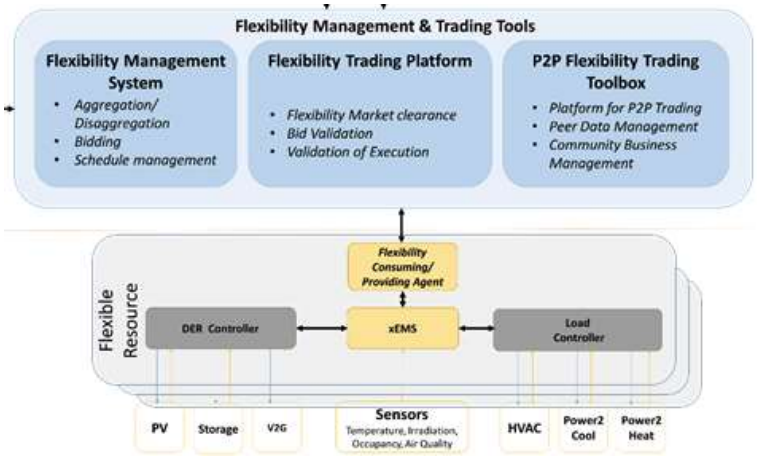

Figure 1 P2P-FTP and related systems from the FEVER overall system architecture

As seen in Figure 1, P2P-FTP is one of the flexibility management and trading tools provided in FEVER. Being a part of the P2P Flexibility Trading Toolbox (WP5), P2P-FTP complements the other FEVER systems from WP2, namely the Flexibility Management System (FMS) and Flexibility Trading Platform (FTP), which, unlike P2P-FTP, offer specialized solutions for *aggregator* users and *centralized trading*, respectively (see D2.3-D2.4). In comparison to these, P2P-FTP offers a stand-alone *decentralized* (P2P) trading solution, as well as provides FMS and FTP access to business-level entities, primarily user *identities* and *assets,* managed by P2P-FTP (through an API) - for FEVER solution cross-breeding and additional overall capabilities offered by the FEVER tools. Conversely, P2P-FTP requires other components (e.g., Energy Management Systems - xEMS, Flexibility Service Providing Agents - FSPA, and/or FMS) to provide some functionality, e.g., for market bid provisioning, data exchange, and control of prosumer assets.

This document summarizes the combined design and specification efforts of Tasks 5.1-5.3. Specifically, it first presents a number of peer-to-peer trading use cases, then focuses on requirements (functional and non-functional), architecture, interfaces, and dependencies of the P2P-FTP, while providing mappings to relevant FEVER use cases and other WP solutions, as well as to other logistics for the successful implementation of WP5.

The document is organized as follows. Section 0 gives simple and intuitive user stories, which explain the process of P2P trading. Section 3 presents high-level requirements stemming from these user stories. Section 4 provides high-level architecture of the P2P-FTP. Sections 5-7 give specifications of the 3 DAPPs of P2P-FTP. Section 0 gives a specification of an external API for integrating P2P-FTP with other FEVER systems. Section 0 overviews the cloud facility as well as continuous integration and continuous delivery pipelines to be used by P2P-FTP. Finally, Section 0 summarizes and concludes this report.

2 User Stories of Peer-to-Peer Trading

In this section, we give simple non-technical user stories, which exemplify the process of peer-to-peer trading from the end-user perspective. Only simplified variants of these user stories will be practically demonstrated in FEVER, by following primarily the FEVER use cases HLUC 15 and PUC32. The use cases presented in this section will set guidelines and directions for the actual development of the P2P-FTP solution. At the end of this section, we will deconstruct these use cases and identify users, assets, and motives, by providing answers to the following questions: “who is trading?”, “what assets are being traded?”, and “why are these assets being traded?”

2.1 US1: Peer buying locally produced electricity from other EC peers

Alice, living in a private house, would like to do her dishes and, for dishwashing, she would like to use electricity produced by PVs (or Battery Energy Storage Systems - BESSes) of her neighbors within the EC. To be able to do that, she needs to make sure she has sufficient credit in her FlexCoin account. She can get FlexCoins, by exchanging Euros, doing some service for the EC (e.g., painting houses), providing some goods (tomatoes, bottles of wine), or allowing the EC to remotely control her household appliances (heater), etc.

Alice activates her dishwasher, which is connected to a smart plug. Alice’s Energy Management System (xEMS) detects the power flow and automatically provides the required amount(-s) of electricity for the full washing cycle of 2-hour duration:

1. EMS automatically generates an electricity buying FlexOffer (FO, a bid) and sends the FO to P2P-FTP.
2. P2P-FTP automatically matches all available seller and buyer bids by exploring *price flexibility* options of Alice, as well as *time*, *energy*, and *price flexibility* options of other peers. Finally, P2P-FTP determines electricity prices and provisions amounts for the full duration of Alice's dish washing. Traded (prescribed) amounts and prices are sent back from P2P-FTP to Alice's EMS, in the form of a FlexOffer schedule.
3. The amount of FlexCoins required for this process is automatically withdrawn from Alice’s account.

While the dishwasher operates, EMS automatically collects measurements from the smart-plug and submits them to P2P-FTP.

During this process, electricity for the dishwasher is supplied by multiple (PVs and BESSes of) neighbors and the rest of the grid in the following way:

- If Alice’s dishwasher consumed electricity amounts **equal to** the contracted (scheduled) amounts for every 15 min interval, the contracted amount is automatically subtracted from the readings of Alice’s main house meter (the connection point) by utilizing the FlexTrading DAPP.
- If Alice’s dishwasher consumed **less than** scheduled – nothing happens. Alice just uses more FlexCoins than she had to (no other penalties applicable).
- If Alice’s dishwasher consumed **more**, the required deficit electricity is provisioned from the grid under the standard grid/supplier tariff (and Alice’s consumption is only partially covered by her neighbors).

As an alternative, Alice also has the option to activate the dishwasher in a delayed manner using a web (or a mobile) APP. When activated through the APP, P2P-FTP automatically finds the best time to buy electricity for the full dishwashing process, giving additional savings to Alice.

2.2 US2: Peer selling PV energy to other EC peers

Bob, living in a household, owns a PV and uses PV-generated power primarily for his own needs. During sunny days, Bob would like to sell up to 50% of PV-produced electricity to his EC neighbors.

Bob configures his Energy Management System (xEMS) accordingly. xEMS automatically monitors the PV inverter sub-meter (when there is one) and makes a prognosis of expected PV yield for the sunny

periods. Based on the prognosis, xEMS automatically sells 50% of the electricity from the Bob's PV to his neighbors:

1. EMS automatically generates an electricity selling FlexOffer (FO, a bid) and sends the FO to P2P-FTP.
2. P2P-FTP automatically matches all available seller and buyer bids by exploring *price* and *energy flexibility* options of Bob, as well as *time*, *energy*, and *price flexibility* options of other peers.
3. Finally, P2P-FTP determines electricity prices and sold amounts. Traded (prescribed) amounts and prices are sent back from P2P-FTP to Bob's EMS, in the form of a *FlexOffer schedule*.

While PV produces electricity, EMS automatically collects measurements from the PV submeter and submits measurements to P2P-FTP. The electricity generated by Bob is supplied to multiple neighbors and the rest of the grid in the following way:

- If Bob's PV produced **at least** the contracted (scheduled) energy amount for every 15 min interval, these contracted amounts are automatically subtracted from the readings of Bob's main house meter (the connection point) by utilizing the FlexTrading DAPP. The contracted FlexCoin amounts are automatically transferred to Bob's FlexCoin account. *Excess* electricity (*actual - contracted*) may be used for Bob's own consumption needs or supplied back to the grid as standard feed-in (as if no P2P-FTP existed).
- If Bob's PV produced **less** electricity than the contracted amount, the payment to Bob in FlexCoins is reduced accordingly based on the cost of (deficit) energy from a back-up energy source (e.g., the grid).

Afterwards, Bob will be able to exchange the earned FlexCoins for other EC values (e.g., fiat currency, wine bottles, etc.).

2.3 US3: Trading between EC peers in the case of a trusted market operator

In this case, there exists a trusted market operator – a locally operating organization, which is trusted by all EC members with the capacity to operate a local flexibility marketplace in a safe and secure manner. It, therefore, uses the centralized marketplace FTP (the FMAR component of the FTP of FEVER WP2) instead of the P2P marketplace (the FlexTrading DAPP of FEVER WP5) to enable trading between EC peers. Since FTP lacks EC-wide user identification and settlement/billing capabilities, the respective functionality of P2P-FTP (the community management/FlexCoin DAPPs) is utilized by the FTP specifically for this. In this case, unlike in US1-2, FTP will perform the actual matching of xEMS/FSPA-generated FOs of the EC members. Still, all FTP's settlement transactions involving FlexCoins will be persisted on the immutable ledger offered by the FlexCoin DAPP. Note, in this case, as it will be explained in Section 7, that the EC will be limited in terms of the flexibility product types supported by FTP, compared to the much broader variety of products supported by (the FlexTrading DAPP of) P2P-FTP.

2.4 US4: Trading directly between a DSO and EC peers

The EC wants to have an operating grid without having to deal with grid calculations and determining load profiles, as well as physical maintenance. Therefore, the local DSO offers such services in return for remuneration, and hence acts as a peer inside the EC.

DSO want to use peers' available flexibility to manage the grid of the EC primarily with their own flexibility. The EMS of the DSO continuously monitors the EC's grid and, in case of events, automatically submits a bid to the P2P-FTP. P2P-FTP automatically matches the DSO bid against those from the EC peers, and automatically transfers FlexCoin amounts from the DSO's to the peers' FlexCoin accounts upon successful delivery of a service.

2.5 US5: Multi-EC trading between a DSO, EC-Operator, and EC peers.

EC is located inside the DSOs grid, but DSO is not taking part in the EC in any way. DSO faces grid stability issues due to low energy production from the PVs in its own grid. The DSO's Energy Management System automatically detects this event and issues a FO into a regional marketplace, enabled by an (P2P-) FTP (WP2 or WP5). In this regional marketplace, both the DSO and the EC-Operator (organization) have access and act as peers. The sent FO is automatically matched with that from the EC-Operator's EMS. The EC-Operator's EMS receives the FO schedule from the regional-level FTP and automatically generates a counter-bid on the local EC marketplace, enabled by P2P-FTP. When matched with the FOs from the EC peers, the xEMSes of the respective EC peers force energy consumption and/or production (i.e., delivery of requested deviations/flexibility) based on the FO schedules received from P2P-FTP. This is followed by the following financial transactions:

1. The regional-level (P2P-) FTP (WP2/WP5) automatically transfers the contracted amounts of $FlexCoins^{REG}$ from the DSO to the EC-Operator account.
2. P2P-FTP automatically transfers the contracted amounts of $FlexCoins^{EC}$ from EC-Operator to the EC peers who have successfully delivered the requested (energy/flexibility) service.

Since the value of FlexCoins ceases outside the EC for which it is defined, FlexCoins used at the regional marketplace ($FlexCoins^{REG}$) might not be compatible with FlexCoins ($FlexCoins^{EC}$) used by EC. In this case, the EC-Operator holds accounts for both FlexCoins variants - $FlexCoins^{REG}$ and $FlexCoins^{EC}$ - the regional and the local variant. To make the conversion between the two variants of FlexCoins, EC-Operator uses a *conversion rate,* equivalent to that used for traditional fiat currencies.

2.6 Summary of the user stories

The table below summarizes the user stories US1-5 by focusing on "what", "who", and "why" aspects.

#	What is traded?	Who is trading?	Why is it traded?
US1	Amount of electricity for a specific appliance	EC member / buyer peer	Use locally produced electricity (save money, reduce CO_2, support EC)
		EC member / seller peer	Sell excess electricity (earn/save money, support EC)
US2	Amounts of excess PV-generated electricity	EC member / buyer peer	Use locally produced electricity (save money, reduce CO_2, support EC)
		EC member / seller peer	Sell excess electricity (earn/save money, support EC)
US3	Flexibility (up/down variations of energy/power)	EC peers	A number of reasons – see FMAR WP2 (D2.4).
US4	Flexibility (up/down variations of energy/power)	DSO	Stabilize the EC grid
		EC members	Reduce EC grid maintenance costs
US5	Flexibility (up/down variations of energy/power)	DSO	Stabilize the DSO grid
		EC-Operator	Bring value to EC (from DSO)
		EC members	Reduce costs

In WP5 (and the remainder of this document), user stories US1-5 will serve as a basis for the overall P2P-FTP design and implementation.

3 P2P-FTP Requirements

In this section, we build on top of the user stories from the previous section and formulate a number of high-level requirements, specific to the P2P-FTP, as well as a number of detailed requirements specific to the pseudo-currency (FlexCoins) and the individual DAPPs of the P2P-FTP. These requirements will guide subsequent P2P-FTP design and specification (Sections 4-0 of this document) as well as P2P-FTP implementation efforts (D5.2-3).

3.1 High-level P2P-FTP (System) Requirements

We now specify a number of high-level requirements stemming from the user stories in Section 0, which underpin the key P2P-FTP functionalities. Some of these will further be detailed in the subsequent subsections.

3.1.1 User roles

Concrete user roles mentioned in US1-5 are summarized in Section 2.6. These, essentially, fall under two categories (roles), peers and EC-Operators. For the overall system configuration and management, we introduce another "admin" role, and define the required types of users as follows:

R1. P2P-FTP needs to support the following three main types of users (roles):

- **EC-Peers** (peers, for simplicity) are EC member users, permitted to use the functionality of P2P-FTP. They may include households, PV owners, EV/storage owners, industries, DSOs, municipalities, as well as existing Energy Communities. Only registered and approved peers are permitted to use P2P-FTP. Their identities and data need to be protected.
- **EC-Operators** are (EC-wide trusted) users that operate and administer an EC: manage *peers*, configure P2P trading parameters, manage FlexCoins accounts, etc. EC-Operators may also represent an EC when trading outside it. EC-Operator is a (trusted) physical EC member/organization.
- **Admin** *users* are (EC- or system-wide trusted) system administrators, developers, testers, etc., which are allowed to monitor the system, rollout fixes, updates, but not directly interfere with EC business processes.

Note, other system-specific user roles/types might co-exist as well, e.g., *endorser, orderer, etc.*

3.1.2 Support for a common user identity system

Since, in the general case, EC members need to deal with several independent FEVER systems (P2P-FTP, FSPA, FTP, xEMS), a platform-wide user identity system is required. It needs to work with multiple ECs and support P2P distributed system scenarios. This need is formulated by the following requirement:

R2. P2P-FTP needs to provide a *common distributed identity system* – allowing to authenticate and authorize users (peers, EC-Operators) from different software systems in FEVER, including the FTP, P2P-FTP, xEMS, FSPAs, etc.

3.1.3 Support for FlexCoins

The need for FlexCoins, the WP5 pseudo-currency mechanism mentioned in US1-5, is formulated as follows:

R3. P2P-FTP needs to offer support for *FlexCoins* – a pseudo-currency mechanism used to *track* and *quantify* (1) commonly owned EC assets and (2) each user's asset contributions, with ability to:

- Exchange FlexCoins with *tangible assets* which have *communal value* to EC
- Exchange asset contributions between users
- Integrate FlexCoins into energy/flexibility trading processes

The requirements for FlexCoins will be further detailed in Section 3.2 .

3.1.4 Support for P2P marketplace

The P2P marketplace which supports scenarios without a trusted market operator and the full range of products mentioned in Section 2.6 is required (US1-2, 4-5). This need is formulated as follows:

> ***R4.*** P2P-FTP needs to provide an online (near) real-time *marketplace* where its participants (peers) are able to trade flexibility and energy products without an intermediary, by utilizing the FlexCoin mechanism (R1, pseudo-currency). The marketplace should be fully *autonomous* and offer *immutability* of records/transactions, also in cases of adversary user behavior.

3.1.5 Support for the machine-to-machine (M2M) Interface

As seen in US1-5, P2P-FTP serves primarily as a *middleware* component used by the technical/operational FEVER systems: xEMS, FTP (also FSPA, FSCA). Therefore, an appropriate M2M interface/API is required:

> **R5**. P2P-FTP needs to offer a machine-to-machine (M2M) interface/API allowing other FEVER sub-systems/tools, including FTP, to access *user identities, FlexCoins,* and the *P2P marketplace.*

3.1.6 Requirement for DAPPs

EC-Operator users (as well as peers) need to be able to monitor and manage peer users, FlexCoin accounts, and P2P marketplace parameters in a simple and intuitive manner. Therefore, P2P-FTP needs to provide some machine-to-human / graphical user interfaces (GUIs), on top of the distributed identify management, FlexCoin, and P2P marketplace implementations (R2-4). With the DLT-based implementations of these in mind, such distributed applications with GUI-based frontends and smart-contract-based backends are often referred to as, simply, DAPPs (distributed applications). This yields the following requirement:

> **R6**. P2P-FTP needs to offer *user identity management, FlexCoin management,* and P2P *marketplace* implementations (R2-4) as the following 3 distinct DAPPs:
>
> - **Community Management DAPP** – this DAPP allows configuring EC values/assets and their members: (de-)registering the types of assets that have communal value to the EC; adding, modifying, revoking users and user access rights, etc.
> - **FlexCoin DAPP** – this DAPP manages FlexCoins. It allows tracking FlexCoin balances as well as FlexCoin transactions, and exchanging FlexCoins with other assets/values within the community (e.g., fiat currencies, bottles of wine, etc.).
> - **FlexTrading DAPP** – this DAPP offers a marketplace where its participants (peers) are able to trade flexibility/energy products without an intermediary. It allows configuring trading parameters (contracts), monitoring and tracking statuses, histories, and states of several parallel P2P markets and individual peers, offering some options for manual interventions/ bidding (e.g., submitting or revoking bids manually from GUI, etc.).

Detailed requirements for individual DAPPs will be specified in Section 3.3.

3.1.7 Support for the deployment environment

From a practical point of view, P2P-FTP requires easy and simplified deployments in new operating environments – considering a, potentially, large number of EC member nodes on which P2P-FTP has to run. This need is formulated as follows:

> **R7.** P2P-FTP needs an effective deployment environment and tools, for simplified (re-)deployments of P2P-FTP in diverse environments of EC members (physical users or organizations).

Next, we further specify and elaborate on some of these high-level requirements.

3.2 Requirements for FlexCoins: P2P-FTP pseudo-currency

In this section, we focus on the requirement R3 and further specify the concept of a pseudo-currency, denoted as FlexCoin. A variant of this concept needs to be natively supported by P2P-FTP and made available as an integral part of the P2P flexibility trading.

In general, FlexCoin should serve as a "tool to track and quantify" each peer's contributions to the EC welfare. The value of a FlexCoin depends on (and is backed by) the value of all tangible assets the EC members contribute and ceases outside the EC which it is associated with.

To track each peer's contribution to the EC welfare, each peer inside the EC must have its private FlexCoin account. The account balance represents the member's share of ownership of all EC assets - in analogy to a company's stock.

The mapping between units of tangible assets (e.g., fiat currency, local tomatoes, bottles of wine) and a single unit of FlexCoin needs to be explicitly specified for the specific EC by the trusted EC-Operator and can, potentially, be customized and changed over time for (some of) the assets and different peer types (e.g., household, industry, PV producer). The two tables below exemplify such a (forward) mapping and its reverse mapping.

Table 1 Forward mapping between the units of *tangible assets* and a single unit of FlexCoin

1 unit [FlexCoin] = XX unit [tangible asset]	**Household**	**Municipality**	**Industry**	**...**
1 Euro	1	1	1	...
1 kWh of locally produced electricity	0.8	1.2	1.4	…
1 ΔkWh/h of on-demand load reduction	0.1	0.2	0.2	…
1 bottle of local wine	0.2	(n/a)	(n/a)	...
1kg of local tomatoes	0.2	(n/a)	0.5	…
1m^2 of houses painted	2	0.5	5	...
...	...	...	...	...

Table 2 Reverse mapping between a single unit of FlexCoin and units of the tangible assets

1 unit [tangible asset] = XX unit [FlexCoin]	**Household**	**Municipality**	**Industry**	**...**
1 Euro	1	1	1	...
1 kWh of locally produced electricity	1.25	0.8333	0.7142	…
1 ΔkWh/h of on-demand load reduction	10	5	5	…
1 bottle of local wine	5	(n/a)	(n/a)	...
1kg of local tomatoes	5	(n/a)	2	…

1m^2 of houses painted	0.5	2	0.2	...
...	...	...	...	...

This mapping informally means that, e.g., an EC member (peer) can contribute with 10 euros to the EC's welfare, and this will appear as 10 FlexCoins in the peer's FlexCoin account. Later, these 10 FlexCoins can be converted into 12.5kWh of electricity produced locally inside the EC.

Additionally, there should exist a common EC account, which holds the units of FlexCoins jointly owned by all EC members (peers). The number of FlexCoins in this shared EC account are managed by EC-Operator and can be reconciled (spent) for the benefit of all EC members using an asset mapping table, similar to the one above.

The balances of all peer FlexCoin accounts together with the balances of the (jointly owned) EC tangible assets need to be tracked using a single common (distributed) ledger. The table below exemplifies the evolution of such a ledger state under 7 distinct transactions Tx1-7 for a single EC (EC1).

Table 3 Example of ledger state evolution for a single EC1 under transactions Tx1-9

# Tx	**FlexCoin accounts (intangible EC assets)**				**Tangible EC assets**			
	EC common FlexCoin account	**Peer 1 personal FlexCon account**	**Peer 2 personal FlexCoin account**	**...**	**Euros**	**Bottles of wine**	**1 kWh of locally produced electricity**	**...**
Initially, EC1 owns no tangible or intangible assets (the genesis DLT block)								
-	0	0	0	...	0	0	0	...
Asset deposit example (Tx 1-2)								
Peer1 contributes 5 EUR to EC1 (in cash or through e-banking). This amount is tracked both as a shared tangible EC asset and as a personal Peer1 FlexCoin intangible asset (with the conversion rate of 1 in this case).								
Tx1		5		...	5			...
-	0	5	0	...	5	0	0	...
Peer2 contributes 2 bottles of wine to EC1, which are converted into 10 units of FlexCoin for Peer2.								
Tx2			10	...		2		...
-	0	5	10	...	5	2	0	
Asset withdrawal example (Tx 3)								
Peer1 exchanges 5 FlexCoins with 1 bottle of wine								
Tx3		-5		...		-1		...

-	0	0	10	...	5	1	0	...
Transferring 3 units of FlexCoin from Peer2 to Peer1 account (Tx4)								
Tx4		3	-3	...	n/a			
-	0	3	7	...	5	1	0	...
EC1 Configuration 1. Buying and selling electricity while generating private, but not communal value for the EC: • **Peer2 buys locally produced electricity/flexibility from EC (Tx5)** • **Peer1 sells PV-produced electricity/flexibility to EC (Tx6)**								
Tx5	2		-2	...	n/a			
Tx6	-1	1			n/a			
-	1	4	5	...	5	1	0	...
These current FlexCoin balances informally mean that: • Peer1 owns 40% (4 / (1+4+5) * 100%) • Peer2 owns 50% (5 / (1+4+5) * 100%) • EC as the whole owns 10% (1 / (1+4+5) * 100%) of all tangible assets contributed by EC members (i.e., 5 EUR and 1 bottle of wine).								
EC common FlexCoin account reconciliation (asset withdrawal) E.g., EC members make a party and jointly drink 1 bottle of wine								
Tx7	-1	n/a				-1	0	...
-	0	4	5	...	5	0	0	...
EC1 Configuration 2 (an alternative to EC1 Configuration 1). Buying and selling electricity by generating communal value for the EC: • Peer2 buys locally produced electricity/flexibility from EC (Tx8) • Peer1 sells PV-produced electricity/flexibility to EC (Tx9)								
Tx8			-2				-1.6	
Tx9		1					0.8	
-	0	5	3	…	5	0	-0.8	

The above example presents the two models (EC configurations) of how energy/flexibility trading can be realized using FlexCoins:

- In the EC Configuration 1 (model 1), electricity/flexibility **is not considered** as a commonly owned EC asset (valuable), therefore Peer1 and Peer2 consuming and producing electricity

does not **mint or burn FlexCoins**. Instead, this triggers EC **asset redistribution**, i.e., exchange of FlexCoins between the private and EC common FlexCoin accounts.

- In the EC Configuration 2 (model 2), electricity/flexibility **is considered** as a communal EC asset (value) and therefore Peer1 and Peer2 consuming and producing 1kWh of electricity triggers FlexCoin (EC asset) **generation (minting)**, i.e., deposits and/or withdrawals of FlexCoins in the respective private FlexCoin accounts. This model is somewhat simpler since no common EC account is required.

A specific model should be selected based on the business configuration of an EC. P2P-FTP will need the support for both models / EC configurations.

3.3 Requirements for P2P-FTP DAPP

In this section, we focus on the 3 DAPPs of P2P-FTP, defined by R6 in Section 3.1.6. We further elaborate on R6, by presenting additional (sub-) requirements from the end-user perspective as simple and intuitive end-user stories. These are attributed with (1) pre-conditions, which define conditions to be met for the user story to be applicable and (2) acceptance criteria, which describe a number of criteria which an implementation of a particular user story has to fulfil. We provide such user stories for each of the P2P-FTP DAPPs in the subsequent sections. The following color scheme and notations are used:

	A story of an *admin* user
	A story of an *EC-Operator* user
	A story of a peer or any other user

→ - pre-condition

- - required functionality

3.3.1 Requirements for the Community Management DAPP

#	User story description	Acceptance Criteria	Priority
SC 1.1	As an *admin* user, I would like to be able to create, update, delete ECs and users, and assign *roles (e.g., admin, EC-Operator, peer)* to different users within a single EC.	→ Authenticated as an *admin* user • Display, edit, delete ECs, users, and roles. • Display, edit, delete, selected user roles in the context of a specific EC.	high
SC 1.2	As an *EC-Operator*, I would like to create, delete, and update *peer users* within an EC where I am the operator.	→ Authenticated as an EC-operator • Display, edit, delete existing users inside the operated EC, and assign user roles.	high
SC 1.3	As a *P2P-FTP user* (peer, EC-Operator, admin), I would like to use the same identity (user account) when using all P2P-FTP services (FlexCoin DAPP, FlexTrading DAPP, FMAN, etc.)	• Provide an API for user authentication and authorization for on-chain and off-chain use.	high
SC 1.4	As an *EC-Operator*, I would like to configure my EC and define EC-specific *types* of peers (e.g., households, PV owners, industries)	→ Authenticated as an EC-operator • Display, edit, and delete existing peer types.	high

3.3.2 Requirements for the FlexCoin DAPP

#	User story description	Acceptance Criteria	Priority
SF 1.1	As an *EC-Peer*, I would like to view my FlexCoin account balance and transaction history so that I know where my FlexCoins came from and went to, and what my current balance is.	→ Authenticated as a peer • Display account balance • Display a list of FlexCoin transactions for a selected period	high
SF 1.2.1	As an *EC-Peer*, I would like to deposit FlexCoins into my personal FlexCoin account by claiming my contributed *tangible assets* to the community, so I could, e.g., participate in P2P trading.	→ Authenticated as a peer • Should be able to create *claims of contribution* (e.g., community service done), which have to be approved by EC-Operator for a specific FlexCoin amount to be calculated and deposited into that account.	high
SF 1.2.2	As an *EC-Peer*, I would like to deposit FlexCoins into my personal FlexCoin account in an automated fashion through an e-banking system, so I could, e.g., participate in P2P trading.	→ Authenticated as a peer • Should be able to calculate amount of FlexCoins to be received for a single unit of a fiat currency (e.g., EUR) - see SP1.5 • Should be able to automatically transfer FlexCoins into the peer's private account (with a corresponding tangible asset record in the ledger, see Section 4).	low
SF 1.3	As an *EC-Peer*, I would like to withdraw FlexCoins from my account, so I could exchange the units of FlexCoints back to my preferred tangible assets (e.g., X units of fiat currency, Y wine bottles, Z kgs of tomatoes).	→ Authenticated as a peer • Should be able to create *requests of withdrawal,* which will have to be approved and fulfilled by EC-Operator.	low
SF 1.4	As an *EC-Peer*, I would like to be able to spend/acquire FlexCoins for buying/selling locally produced electricity/flexibility from/to other peers within an EC	• Provide an API for on-chain and off-chain FlexCoin transactions. • Permit (trusted) trading applications (FMAR, FlexTrading DAPP) to transfer amounts of FlexCoins from one user (peer) account to another using an API.	high
SF 1.5	As an *EC-Operator*, I would like to be able to configure a mapping between a *single unit of FlexCoin* and units of *tangible assets* for different peer types in order to specify the value of FlexCoin for different *peer types* within my operated EC (see Section 4).	→ Authenticated as an EC-Operator • For a selected EC and a peer type, allow defining a mapping between a single unit of FlexCoin and all	high

		tangible assets, specified in SC 1.4	
SF 1.6	As an EC-Operator, I would like to see all (pending) user *claims of contribution* (SF1.2) and either *reject*, accept or *correct and accept* them. On acceptance, I want that a specific FlexCoin amount will be automatically calculated based the mapping function and on the quantities of tangible assets a peer has contributed, and automatically transferred to the peer's personal account, while also tracking the units of tangible assets contributed by the peer.	→ Authenticated as an EC-Operator • Allow seeing, (creating), rejecting, correcting, and accepting all pending *claims of contributions*	high
SF 1.7	As an EC-Operator, I would like to see all (pending) FlexCoin *requests of withdrawal.* If I am able to fulfill this request (i.e., have all requested tangible assets in stock), I want to be able to confirm this request and automatically withdraw a specific amount of FlexCoins from the peer's FlexCoin account.	→ Authenticated as an EC-Operator • Allow seeing, (creating), rejecting, correcting, and accepting all pending *requests of withdrawal.*	low
SF 1.8	As an EC-Operator, I would like to own and manage the EC common FlexCoin account.	→ Authenticated as an EC-Operator • Allow EC-Operator managing its FlexCoin account in the same way as peers do.	high

3.3.3 Requirements for the FlexTrading DAPP

#	**User story description**	**Acceptance Criteria**	**Priority**
ST 1.1	As an *EC-Operator*, I would like to enable and disable trading of different energy flexibility products and configure a P2P trading marketplace within my EC so I can manage the trading process	→ Authenticated as an EC-operator • Display the current status of the P2P trading marketplace, enable/disable certain flexibility products.	high
ST 1.2	As an *EC-Peer*, I would like to be able to generate requests for *joining* or *leaving a* peer-to-peer marketplace, to be able to opt-in or opt-out of P2P trading. When joining, I agree to share my power consumption/production data from specified (sub-)meter(-s). Note, this relates to UI_P2P_02 (D1.2): "I want to have the right for opt-in and opt-out of trading so I have full control over my participation as a peer in the P2P trading."	→ Authenticated as a peer • [Through GUI and API], allow (an xEMS of a) peer generating *requests for joining/leaving;* The following details must be provided when joining: a (sub-)meter details (HW ID, etc.), type of device connected.	high
ST 1.3	As an *EC-Operator*, I would like to be able to *approve* or *decline* the peer's participation	→ Authenticated as EC-Operator • Allows seeing all approved and	high

	for a particular peer, if I suspect some malicious peer behaviour.	pending peers' requests • Allow accepting or *revoking* peers' *requests for joining*.	
ST 1.4	As an *EC-Peer*, I would like my xEMS to be able to automatically provision electricity/flexibility from other peers in my EC and/or offer my energy flexibility based on my individual settings (e.g., comfort constraints).	→ Authenticated as peer • Peer is *approved* (ST1.3) • Offer API which will allow peer's xEMS submitting FlexOffers (FOs), FO updates/revocations, and metered data from the *approved peer's* meters, and receiving FO schedules, electricity, flexibility prices in FlexCoins. • Offer on-chain FO matching mechanism to match peer offers and requests • Offer automatic FlexCoin transactions, which follow FO execution.	high
ST 1.5	UI_P2P_05 (D1.2) As an *EC-Peer*, I want to have access to the visualization of all my trading results so that I know if my bids and requests are traded and how economic it is. UI_P2P_06 (D1.2) I want to have access to the visualization of the contribution of each of my assets in terms of (number of flexibility offers/requests, duration, money earned).	→ Authenticated as peer • Display the history of peer's FOs, FlexCoin transactions, and KPIs for a selected time period.	high
ST 1.6	[UI_P2P_01 - D1.2] As an *EC-Peer*, I want to have access to the visualization of the timeline of flexibility offers and requests within my community to learn how our flexibility affects the behaviour of the whole energy community (EC). This way, I can see how our joint participation increases self-consumption of my EC.	→ Authenticated as a peer • Display offers/requests, summaries, and KPIs for a selected time period of all peers within my EC.	low
ST 1.7	[UI_P2P_04 - D1.2] As an *EC-Peer*, I want to receive an automated notification whenever an asset or my EMS is not available for participation in trading so that I can check on its status.	→Authenticated as a peer • Display the current peer status: number of active FOs, last available measurements from a meter, KPIs, etc.	low
ST 1.8	As an *EC-Operator*, I would like to see actual electricity amounts measured by the approved peer (sub-)meters, as well as actual energy flows between the peers during their trading periods. I will use this information to prove to the EC's external electricity supplier so electricity amounts traded in the P2P marketplace are excluded (not accounted for) in the final EC electricity bill.	→ Authenticated as a peer • Display power readings and consumed/produced energy amounts of the complete EC and/or individual peers by aggregating measurements from some or all approved (sub-)meters.	high

4 P2P-FTP Architecture and Interfaces

In this section, we build on top of the DAPP requirements, and aim to present a high-level P2P-FTP architecture and P2P-FTP interfaces exposed to other FEVER components (primarily WP2). First, we present the conceptual architecture of P2P-FTP, followed by an overview of the selected blockchain technology. The in-depth specifications of the DAPPs are provided in the subsequent sections.

4.1 High-level architecture of P2P-FTP

As formulated in R6 (Section 3.1.6) the P2P-FTP consists of 3 inter-linked DAPPs – Community Management DAPP, FlexCoin DAPP, and FlexTrading DAPP. These are distributed applications running on top of the Hyperledger Fabric (HLF), with a number of inter-linked software services (components) of the following types:

- ***Front-end* services** – expose graphical user interfaces (GUIs) to end-users and handle communication between end-users and other back-end and HLF services.
- ***Back-end* services** – encapsulate all off-chain logic of the DAPPs. They expose internal *application programming interfaces* (APIs) with a number of (HTTP) end-points, which can be accessed directly by the front-end services and indirectly (via API for Integration Services) by external applications. These back-end services also act as HLF *client applications*, which interact with the blockchain network by generating and submitting transactions, and/or querying the ledger content.
- **HLF *network* services** – are HLF-specific services, which are required to run the network. These include *HLF (endorsing and/or committing) peers*, *orderer services*, *smart-contracts, channels definition* and *certificate authorities*. Among these, smart-contracts encapsulate on-chain-specific logics of the DAPPs.

Instances of these DAPPs (services) run in a single (multi-organization) cloud (deployment) environment, which isolates these services from a direct external access, by acting as a sandbox. End-user client (i.e., a web browser) as well as external applications interact with P2P-FTP only through a reverse proxy (e.g., Nginx, HAproxy), which redirects all incoming requests to specific DAPP services. Specifically, external applications (other FEVER systems) communicate with P2P-FTP through an exposed so-called API for Integration Services, which expose certain (a limited number of) back-end and blockchain service functionalities to other FEVER systems using a simple REST interface. It is envisioned that energy management systems (xEMS, FMS) used by different types of users (EC-Operator, prosumer peers, aggregator peers, DSO peers) will be equipped with specialized flexibility service consuming/producing agents (FSPAs/FSCAs) which will handle automatic exchange of data between xEMS and P2P-FTP via this exposed API (R5 – Section 3.1.5). A special case is FTP (centralized, WP2): it will use the exposed API for communication with FlexCoin and Community Management DAPP services, but not with the services of the FlexTrading DAPP – since FTP itself offers a standalone marketplace system (centralized as opposed to P2P).

To ease the (re-)deployment of P2P-FTP services (R7, Section 3.1.7), a continuous integration and continuous deployment (CI/CD) pipeline will be utilized. This CI/CD pipeline, as well as the P2P-FTP DAPPs, and the API for Integration Services will be specified in detail in the indicated sections (see Figure 2).

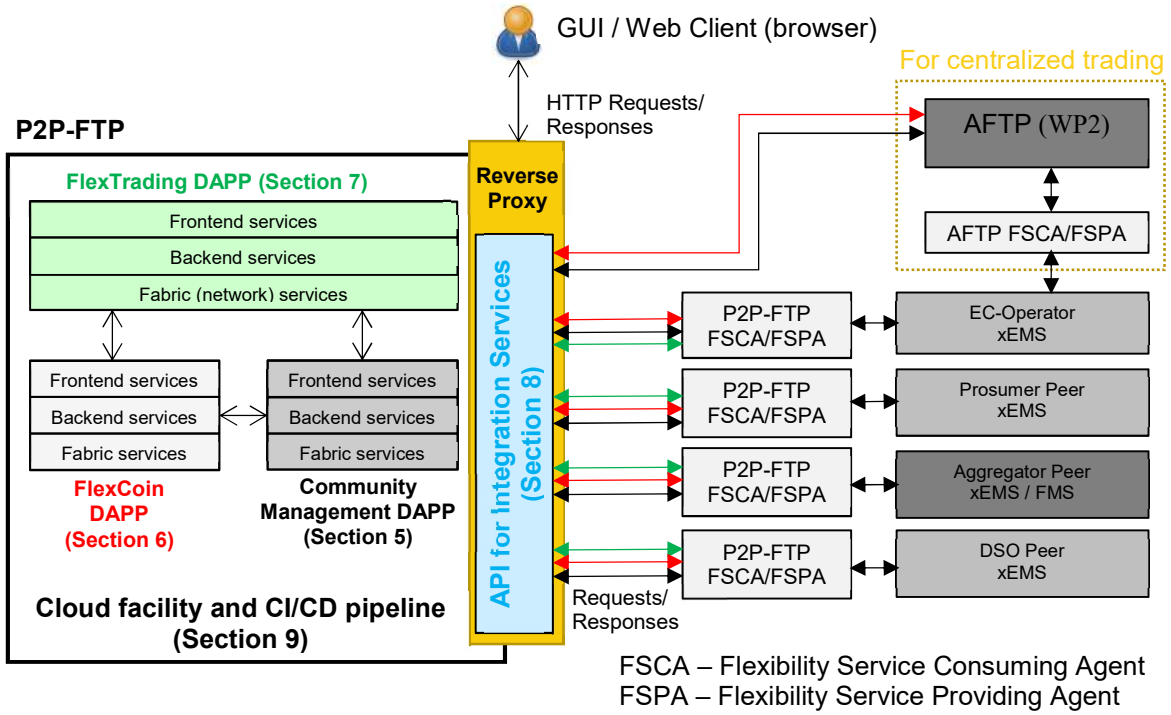

Figure 2 High-level P2P-FTP Architecture

Since P2P-FTP builds up on top of the Hyperledger Fabric, special attention is given to this blockchain technology in the next sub-section.

4.2 Hyperledger Fabric Overview

Hyperledger Fabric (HLF) is a modular and extensible open-source system for deploying and operating permissioned blockchains that was originally developed by IBM, and it is one of the Hyperledger projects hosted by the Linux Foundation [3]. Fabric is designed to be extensible and flexible to accommodate different needs and use cases. As such, it supports modular consensus protocols and authentication mechanisms, which allows the system to be tailored to different use cases and trust models. Fabric is also the first blockchain system that runs distributed applications (DAPPs) written in standard, general-purpose programming languages, without systemic dependency on a native cryptocurrency. Notably, Fabric realizes the *permissioned model* for membership management, unlike public blockchain protocols, such as Bitcoin [4] and Ethereum [5]. In terms of scalability, HLF currently outperforms Ethereum and Bitcoin by two orders of magnitude [6]. Since the release of version 1.2 (2018), the framework support so-called *private data collections* which allow to keep certain data and transactions confidential among a subset of blockchain participants – a feature which is particularly relevant for P2P-FTP [7].

4.2.1 Architecture

The architecture of HLF follows a novel *execute-order-validate* paradigm for distributed execution of untrusted code in an untrusted environment. It separates the transaction flow into three steps, which may be run on different entities in the system: *execution* that is executing a transaction and checking its correctness; *ordering* transactions through a consensus protocol semantics; and transaction *validation* per application-specific trust assumptions. This design departs radically from the traditional *order-execute* paradigm in that Fabric typically executes transactions before reaching final agreement on their order.

4.2.2 Transactions, consensus protocol

The HLF is a distributed system consisting of a number of nodes that communicate with each other. These nodes are the communication entities of the blockchain. A “node” is only a logical function in the

sense that multiple nodes of different types can run on the same physical server. A Fabric blockchain consists of a set of nodes that form a network. As Fabric is permissioned, all nodes that participate in the network have an identity, as provided by a modular membership service provider, nodes in a Fabric network take up one of three roles: *clients (aka., client applications)* submit transaction proposals for execution, help orchestrate the execution phase, and, finally, broadcast transactions for ordering; *peers* execute transaction proposals and validate transactions; and, *orderers* are the nodes that collectively form the ordering service, that establishes the total order of all transactions in Fabric.

The HLF runs programs called *chaincode*, holds state and ledger data, and executes transactions. The chaincode is the central element as transactions are operations invoked on the chaincode. Chaincodes encode the application logic into electronic form that is executed through the transactions. Each chaincode may define its own persistent entries in the state. Transactions have to be endorsed and only endorsed transactions may be committed and have an effect on the state. There are system chaincodes for management functions and parameters.

In the latest main release of HLF v2.x, a new feature towards the democratization of the system was introduced. The new Chaincode Lifecycle [8] allows Consortium members to agree on a chaincode change before its logic is actually deployed on the channel. This process prevents any possible unexpected behaviour caused because of non-installed chaincode on all peers.

4.2.3 Channels

A Fabric network actually supports multiple networks, which can be considered as separate stand-alone blockchains that are connected to the same ordering service. Each such blockchain is called a *channel* and may have different peers as its members, for the purpose of conducting private and confidential transactions. Channels can be used to partition the state of the blockchain network, but consensus across channels is not coordinated and the total order of transactions in each channel is separate from the others. Peers can belong to multiple channels, and therefore maintain multiple ledgers, no ledger data can pass from one channel to another. Ordering of the transactions is established per channel.

The aforementioned concepts of a peer, ledger, client and their inter-dependencies are depicted in Figure 3.

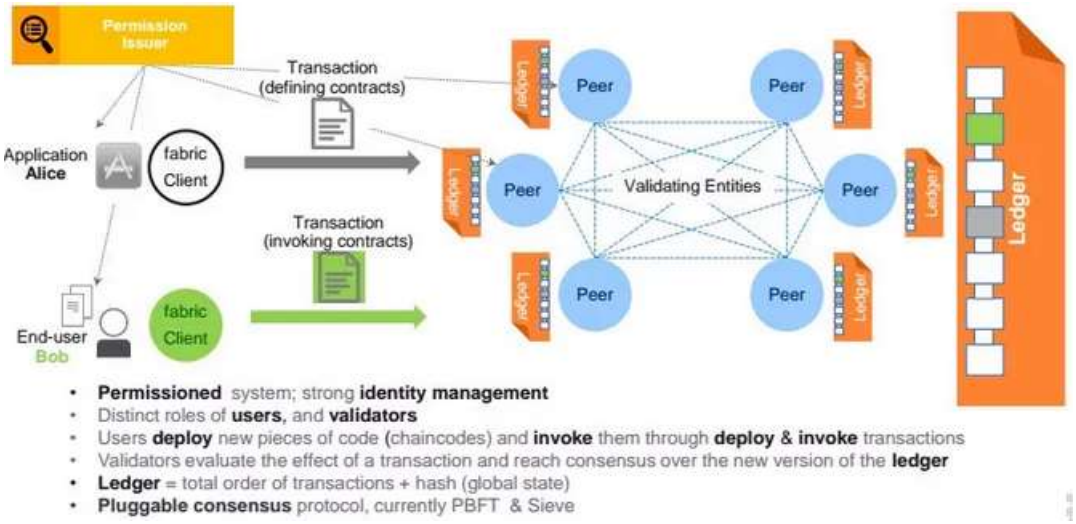

Figure 3 Hyperledger Fabric Model (from [9])

5 Community Management DAPP Specification

In this chapter, we begin by introducing fundamental concepts that are related with the issue of digital identity in the context of HLF, its corresponding management lifecycle and the means under which those can be adapted in the context of ECs, according to the requirements of Section 3.3.1. While HLF has a variety of attractive features regarding digital identity management, i.e., modularity and flexibility, which also extends to concepts that are built on top of it, such as access control and authorization, it simultaneously introduces challenges in terms of its end-to-end configuration complexity. For instance, there are multiple layers of access control, related to arbitrary policy hierarchies that network architects can define in the context of a particular organization, to channel-wide ones regarding various important aspects of, e.g., a channel's creation, such as endorsement, modification, read and write policies, just to name few. In light of this fact, it is preferable to abstract all these intricacies of HLF in simplified APIs that can be employed by, e.g., traditional web developers. The benefits of this approach are numerous. Blockchains are a niche topic and it is advisable to abstract their inherent complexity from developers that are not accustomed with this technology to, on the one hand, reduce errors and, on the other hand, facilitate integration with other web services that are developed, first, in the context of WP5 and, subsequently, with those of WP2.

5.1 Federated Identity Model

HLF is an enterprise-grade, permissioned blockchain platform which allows multiple administrative domains (organizations) to pool together their distributed infrastructure to create a trusted, secure and verifiable transactional framework. In this constrained setting, individual organizations need to be able to issue digital identities for entities that are part of their respective domain, e.g., a company should be able to issue digital identities to its employees. Moreover, these digital identities need to be verifiable by other organizations that are part of the consortium. Put simply, an organization should be able to infer the organization to which the entity that it is communicating with belongs to. In the remainder of this section, we provide an overview of the federated identity model, which is employed by HLF.

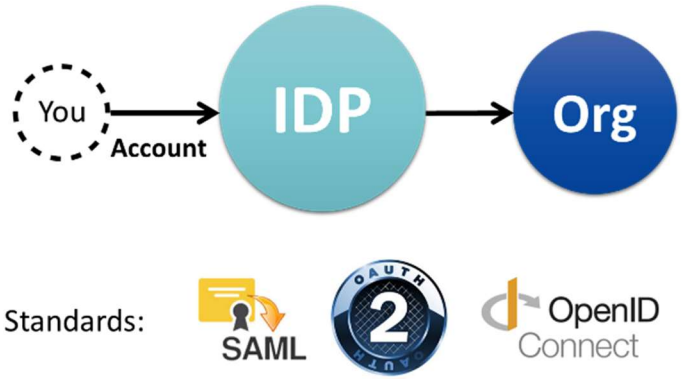

Figure 4: Illustration of the three-way relationship that is intrinsic to the federated identity paradigm, along with technical standards that are associated with it.

The underlying principle of the federated identity model is straightforward; an identity provider (IDP) is inserted as an intermediary between an entity ("You") and an organization ("Org"), as illustrated in Figure 4. Entities have only one identity account with the IDP in this paradigm, which lets them log-in and share basic identity data with any website, service, or app. A federation is a vague term that refers to a collection of services, e.g., websites that share the same IDP (or even group of IDPs). The dependent parties in a federation are the organizations that participate in it (RPs).

Since 2005, three generations of federated identity protocols have been developed with great success: Security Assertion Markup Language (SAML), OAuth, and OpenID Connect. Thanks to these protocols, single sign-on (SSO) is now a standard feature of most company intranets and extranets. The term "user-centric identification" was used to describe federated identity management (FIM) on the consumer Internet. Thanks to protocols like OpenID Connect, social login buttons from Facebook, Google, Twitter, LinkedIn, and other companies are now a standard feature on many consumer-facing websites (Figure 5). Focusing specifically on HLF, an organization's IDP is referred to as membership service provider

(MSP) which, in Internet standard terms, corresponds to a certification authority (CA), or hierarchy thereof, that allows organizations to create, sign, and issue public key certificates, following the X.509 standard [10]. A brief overview of an organization's CA (hierarchy) functionalities is as follows:

- Establishment and maintenance of a certification policy (CP).
- Digital certificate issuance and revocation, according to the specifications of its CP.
- Acting as a root of trust, or a trusted third party, that ensures the binding between an entity's public key and the data encoded in the corresponding public key certificate.
- Provisioning secure storage and management processes in respect to the usage of its signing key.

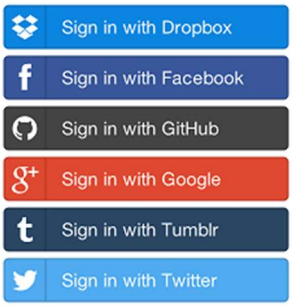

Figure 5: SSO buttons that appear prominently on websites in an attempt to simplify registration and login processes.

5.2 Privacy

In today's societies, much domestic legislation recognizes privacy as a basic right. This phrase is perhaps fundamentally abstract and devoid of specific meaning, a problem that this section aims to remedy. The problem derives from the fact that the definition of privacy has evolved through time, owing to the impact of new technology. In terms of privacy, FEVER's purpose is to preserve the privacy of individual EC members, the EC operator, and other ecosystem stakeholders.

The project's requirements, as outlined in WP1, stipulate that each EC member has no access to data pertaining to other EC members. This includes information that can be used to directly or indirectly identify another EC member, as well as information about measurements reported on assets deployed on EC members' premises. We emphasize that the EC operator maintains access to the personal/private data of community members via the FEVER platform because both parties have signed a legally-binding contract, i.e., the EC member has given his approval via a real-world contractual arrangement, and so privacy is not breached. Other entities cannot access these data points, despite the fact that the EC operator has complete access to them. When such parties need to process sensitive, personal data, the EC member's consent will be sought, and once granted, the needed data will be made available to these parties through the EC operator's infrastructure. Lastly, it is imperative that EC members do not have access to data that are relevant to the EC operator's internal operations and processes. This is a necessity to ensure that the privacy of the EC operator's business logic is preserved.

Additional privacy requirements, as documented in D1.2, relate to congruent concepts, such as access control, usability, interoperability, minimal dependency on user's memory in regard to secret management, and others. Based on the aforementioned, as well as, the requirements of Sections 3.1.1, 3.1.2, and 3.3.1, in the following, we provide an intuitive overview of appropriate mechanisms to address them all in the context of FEVER's federated identity model.

The Role-Based Access Control (RBAC) mechanism is one of the well-described access control mechanisms in the literature. This method is based on the concept of roles and the groups of users who are assigned to those roles. The roles indicate various duties that must be completed, and the users are assigned authorization to the tasks as well as restrictions on access to these roles (tasks). Roles' permissions and user groups' permissions can both change. On the one hand, a role represents a group of people, while on the other, it represents a group of permissions.

RBAC encompasses the ideas of roles, hierarchies, role activation, limits on user/role membership, and

role set activation in an access control architecture. When it comes to simplifying security policy management, RBAC is a smart option. It's a technique that's seeing an increase in the number of vendors who provide it.

RBAC core is a collection of RBAC's fundamental ideas. Users are assigned to roles on the one hand, while permissions are assigned to roles on the other. Users, on the other hand, obtain permissions as members of roles. Users and roles should be clearly defined. Roles can be selectively activated and deactivated using the idea of user sessions. The characteristics of group-based access control are applicable to the RBAC base model in general. The interconnections between the RBAC core elements are depicted in Figure 6.

The arrows in Figure 6 entail that multiple roles can be assigned to one user and vice versa; the same goes for permissions and role assignments. Permissions are exercised by users. The session stands for the assignment of many roles to one user, i.e., a user sets a session where some of the roles are activated from the total set of roles that this user is assigned to.

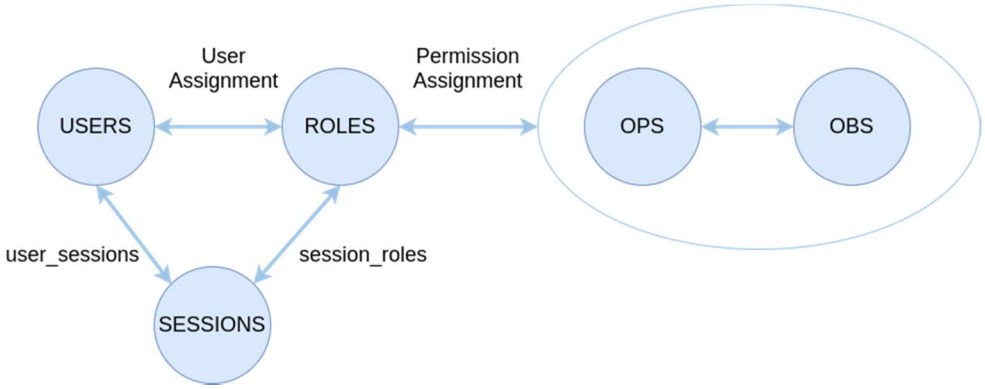

Figure 6: RBAC's primary components and their interrelationships

Regarding the functional specifications of RBAC core, the functions needed to be dealt with are: administrative functions, supporting system functions and review functions. Administrative functions revolve around the creation of element sets. For instance, users and roles are created and deleted by the administrator, i.e., the EC operator in FEVER's case, who is also responsible for setting the relationships between roles, operations (OPS) and objects (OBS). The main relationships to be defined are role-to-user and role-to-permission. Supporting system functions revolve around session creation and deletion. The review functions provide the necessary interfaces to review the results of the actions created by the administrative functions.

Table 4: Overview of the main characteristics of RBAC signifying its appropriateness in FEVER.

Characteristic	RBAC
Adaptability	√ (up to moderate-sized user sets)
Scalability	√
Simplicity	√
Simple rule support	√
Complex rule support	√
Dynamically parameterized rule support	X
User permission customization	Requires the establishment of new roles

Granularity	Low

In Table 4, we provide an overview of the main characteristics of RBAC signifying its appropriateness in the context of FEVER. Indeed, RBAC is suitable for up to moderate-sized user sets, which is the case for ECs. Furthermore, the concept of granting the same access credentials to a group of users that share the same function is central to its design process and abides by requirements SC 1.1, SC 1.2 and SC 1.4 (see 3.3.1). For instance, in the case of FEVER, EC peers are assigned the same role that allows them access to the same set of resources, in terms of the infrastructure provided. To ensure interoperability and standards-compliance, these roles will be encoded in the X.509 digital certificates of FEVER's users.

The smart contracts that define the business logic of the DAPPs have an extra access control layer with a greater granularity. When a smart contract is invoked, it is feasible to examine the caller's identity and utilize that information to determine the workflow to be followed. The HLF Client Identification Chaincode (CID) library allows the invoking party's identity to be retrieved. The client's identity can be verified against its organization's Membership Service Provider (MSP), as well as the client's particular roles, which are stored in its certificate and can be obtained and utilized to make access control choices. This provides for a standardized and interoperable mechanism for enforcing RBAC-based access control. We refer the interested reader to a future section of this deliverable where the same constructs are employed to mediate access control at the network layer of HLF.

5.3 Architecture Overview

From here on in, we introduce, in gradual detail, an infrastructure that is comprised by a collection of services that deliver a secure federated identity solution that has a variety of features. First, simplicity of usage, both from the perspective of end-users (EC peers), as well as administrators of the EC operator. This property will, in addition, provide for a less error-prone integration process with other web services that are part of, not only WP5, but WP2 as well. Second, to provide for interoperability, we build on top of industry-wide security standards for authentication and authorization, such as OAuth 2.0 and its popular extension OpenID Connect. In a few words, we essentially build a bridge between these two aforementioned standards and that of X.509 public key certificates in such a manner that allows us to address SC 1.3, i.e., users, regardless of their role, have a single digital identity across the entirety of the FEVER's platform. This encompasses interactions with both off and on chain interactions that users engage in.

In Figure 7, we provide a high-level, architectural overview of the community management DAPP service ecosystem, the interactions amongst its constituent components and their corresponding purpose, as well as the communication protocols employed. In the remainder of this section, we provide preliminary descriptions of all of these services that will facilitate the reader in gaining an intuitive overview of this ecosystem of services. We then dedicate separate sections for each of these services where we engage in more concrete technical analysis and specification.

We begin our description starting from the left-hand side of Figure 7, i.e., the community management DAPP itself. This service is comprised by two parts. First, a secured backend that allows administrators of the EC operator to manage, via simple and powerful APIs, the digital identities of EC members. In a later on section, we provide more concrete details in regard to the envisioned flows, however, at this point, it suffices to say that the administrator will be able to have a concrete view of all the members that have been admitted in FEVER's platform. We envision that this will involve some kind of personally identifiable information (PII) of EC members which, as of yet, have not been concretely defined. The administrator will also be able to remove, or revoke the credentials of admitted EC members, which will result in them being unable to interact with any of FEVER's services. Furthermore, the administrator will be able to examine registration applications of EC members that wish to be on-boarded on the EC's platform and decide accordingly. These functionalities are exposed via a user-friendly UI, which is the second component of this service. The community management DAPP, on a more technical note, can be thought of as an external identity provider (IdP) that, apart from the aforementioned functionalities, provides for login and logout services that involve the issuance of JWT tokens, following the

specifications of OAuth 2.0 and, when required, OpenID Connect. These tokens are employed by edge software components, such as web browsers, for authentication and authorization purposes when interacting with the other components of this ecosystem, which we discuss below.

The next component in line is the HLF Organization CA Service which, in a few words, is a standard certification authority that exposes a set of expressive and configurable endpoints that allow the lifecycle management of X.509 digital certificates, pertaining to a specific organization of a HLF network, e.g., that of the EC operator. The endpoints of this service are restricted to administrators of the EC operator, which are in turn authenticated and authorized via the functionalities of the community management DAPP. This service exposes registration and enrolment endpoints, which are invoked as part of a user's registration process via the community management DAPP and involve the generation and signing of X.509 digital certificates that internally encode the attributes pertaining to the admitted user's role. Moreover, as illustrated in the figure, this service interacts with an HLF network via the gRPC protocol to issue MSP configuration updates, which are essential when, e.g., a user's X.509 digital certificate is revoked as part of her departure from FEVER's ecosystem. We discuss this process in more detail in the corresponding section that is dedicated to the technical specification of this service.

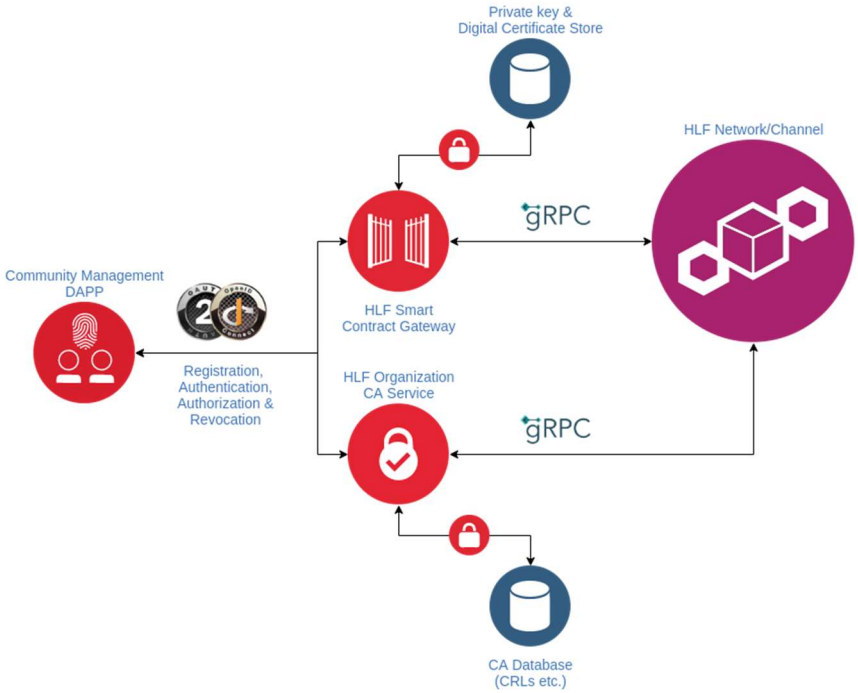

Figure 7: Architecture overview of the community management DAPP service ecosystem and its involved interactions among complementary services.

The services described up to this point are sufficient to cover all of the digital identity requirements of both WP1, as well as, of Section 3.3.1. Indeed, on the one hand, they provide for industry standard means of user authentication and authorization and, on the other hand, expose the necessary means for managing the lifecycle of X.509 digital certificates for all actors involved in FEVER's ecosystem. However, based on the description provided so far, it is arguably the case that these two mechanisms, i.e., the one based on JWTs and the one based on X.509 public key certificates are disconnected. Hence, there is an inherent need to bridge this gap in order to map a user's account maintained in the Community Management DAPP to the corresponding digital certificate when it comes to the user's interactions with the ledger. And more specifically, with the smart contracts that are instantiated on the HLF network. All of the aforementioned issues are addressed by the HLF smart contract gateway which, as a starting point, exposes an authenticated API that allows the EC operator to manage user credentials that are issued by the CA service, i.e., private keys, digital certificates and MSP identifiers. These credentials are stored in a secure database and are employed when an authenticated user wishes to interact with the on-chain smart contracts. This leads us to the second functionality of this microservice, which is an abstraction of all the involved intricacies of interacting with smart contracts of an HLF network. Put simply, this gateway service is able to connect to multiple HLF networks (or channels) and

provide an HTTP façade of the functions exposed by smart contracts in a one-to-one fashion, i.e., each smart contract function is exposed as its own distinct HTTP endpoint. Based on the JWT authorization token, this service is able to infer the invoking user's credentials by appropriately querying its secure database and employing them for digitally signing transactions.

5.4 HLF Organization CA Service

Up to this point, we have provided a high-level, intuitive description of the mechanisms exposed by the EC operator's community management DAPP that allow administrators to onboard or revoke credentials of EC members that, in turn, allow them to interact with FEVER's off-chain web services. However, since HLF is a permissioned blockchain platform that follows the federated identity model (Section 5.1) by building on top of X.509 public key certificates, users (regardless of their role) need to be provisioned with such digital certificates that will allow them to prove their identity to HLF's peer and ordering nodes when issuing transactions to the network.

HLF introduces the concept of Membership Service Providers (MSPs) which, in a nutshell, is an abstraction for the underlying mechanism that allows network entities to verify identity proofs of each other. From a more practical standpoint, an organization's MSP corresponds to a certification authority (CA), or an arbitrary certification hierarchy in more elaborate deployment settings. In addition, to provide for privacy and secure communications, HLF network organizations specify a separate TLS CA. Best practices suggest that the MSP and TLS CAs of an organization are distinct for security and resource isolation purposes. The self-signed X.509 public key certificates of these CAs of each organization that comprise a channel's consortium are encoded in its corresponding configuration transaction(s).

The definition of an HLF organization in the context of a channel can be further divided into one or more organizational units (OUs), which provide the means to structure, or compartmentalize an organization's definition further. Each of these OUs can be associated with a different MSP which, on the one hand, provides for increased fault-tolerance and isolation, similar to the ones provided by the separation of MSP and TLS CAs and, on the other hand, for defining more fine-grained access control policies across all layers of a channel's definition, as well as, at the level of smart contracts.

A special kind of OU, which is referred to as "Node OU", is reserved by HLF to convey the digital certificates of the MSP(s) for network roles that are inherent to an HLF channel definition, i.e., peer and orderer nodes, as well as, organization clients and administrators. For instance, in the context of FEVER, EC members correspond to clients of the EC operator's organization from the perspective of the network's definition. The digital identities of an organization's administrators are of particular importance since they are involved in cases where channel configuration updates are warranted. Discussing all types of channel configuration updates is out of the context of this discussion, apart from one which we discuss in the following.

An important characteristic of any production-grade network, such as HLF, is its ability to adapt, at run-time, to events that affect its initial configuration. More specifically, in the context of an organization's MSP, the network must be able to tolerate cases where, e.g., clients depart from an organization, (intermediary) MSP or TLS CAs are compromised, and others. Put simply, the network needs to provide organizations a mechanism that allows them to, on the one hand, revoke credentials of its members. Clearly, in such events, other organizations that are part of the channel's consortium definition should be notified to maintain the network's security. To address such scenarios, the administrators of organizations are able to digitally sign and broadcast to the channel configuration updates that reflect updates to their organization's MSP structure. In most cases, these revolve around certificate revocation lists (CRLs), which are standardized data structures that convey the subject key identifiers (SKI) of X.509 digital certificates that should no longer be considered valid by any organization in the channel.

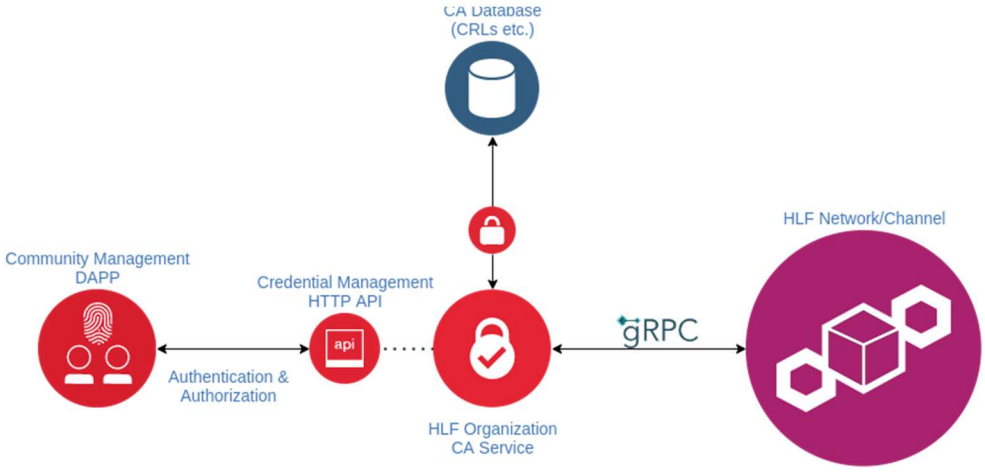

Figure 8: The HLF Organization CA service exposes a credential management API that is restricted to administrators of the EC operator via its integration with the community management DAPP.

The goal of this section is to present the design of a service that, on a high-level, can act as an organization's MSP and TLS X.509 public key certificate provider. One of our fundamental design goals is to completely abstract the intricacies that were previously highlighted from the remainder of FEVER's service ecosystem and its development process. On the one hand, this will provide for ease of integration and, on the other hand, it will minimize the potential for a large variety of security-related errors in the context of X.509 public key certificate lifecycle management, which can prove to be catastrophic to the underlying HLF network security.

The starting point of any HLF network organization revolves around bootstrapping its MSP and TLS CAs. This involves the generation of the private signing keys and their corresponding self-signed X.509 public key certificates that are part of the service's initialization process. This service exposes endpoints that allow the retrieval of these certificates in order to facilitate the, subsequent, establishment of channel configuration transactions, as well as, MSP configuration files that contain the definition of the organization's Node OUs. Moreover, it internally generates and maintains credentials related to the particular's organizations administrator which, for security purposes, are not exposed and are maintained in its secure data store. These are utilized in the context of MSP configuration updates, a process which we present more concretely in a future section of this chapter.

The focal part of this service's credential management API revolves around endpoints that allow administrators of the EC operator to register and, subsequently, enrol digital identities for all roles that are recognized by HLF. During registration, the EC operator's administrator is able to encode additional attributes in the entity's certificate that can be related to role types of FEVER's ecosystem, e.g., EC peers. The output of this registration process is a secret that can be used for enrolment, which is the process that, put simply, generates the private key and the corresponding X.509 public key certificate, both of which are returned as a part of the service's response. The separation of these two processes allows the integration with potential edge agents that can maintain their own self-sovereign credential storage but do not have access to a proper entropy source, which is of grave importance when generating cryptographic material. Although support for such settings is provided, these are considered out of scope of FEVER's architecture. Moreover, again for each HLF role, the service exposes endpoints that unify these two processes for the sake of simplicity. We stress that this service will return an error in cases where an attempt is made to generate a private key and digital certificate pair for an entity that is already successfully registered.

Lastly, this service's API exposes an authenticated endpoint that allows administrators of the organization (e.g., EC operator) to revoke the digital certificate of any of its members, such as EC peers. To convey this fact to the other organizations that are part of channel's consortium, this service employs its internally maintained administrator digital credentials to formulate and digitally sign a CRL, which is wrapped in an MSP configuration update, sealed in an update envelope and broadcasted to the network. Once this transaction is successfully published as part of a new block, the credentials of the revoked entity will be considered invalid by the entirety of the channel and, hence, access to the network's services will be denied by the HLF nodes (peers, orderers).

5.5 HLF Smart Contract Gateway

On a high-level, FEVER's platform is comprised by two independent collection of services, or software components. On the one hand, there is a large variety of off-chain software components, such as the ones that are developed in the context of WP5, that predominantly employ the REST paradigm for their interactions. On the other hand, there is the "blockchain world", i.e., FEVER's, HLF-based, blockchain infrastructure that hosts an ecosystem of digital agents (smart contracts), some of which are introduced in future sections of this deliverable, which is maintained by multiple and distinct administrative domains. Actors that are part of FEVER's ecosystem interact with these digital agents by posting appropriately formed transactions that are, in turn, received by the network's peers and processed accordingly. Arguably, and as was hinted earlier, integration between off and on chain components is an important issue that we dedicate this section to address it.

One way to solve this integration problem, from a software development point of view, would be to import one of HLF's SDKs directly in the codebase of the (backend) services that interact with the ledger. This technique introduces an incompatibility, in regards to the programming languages that are used for the component's development. This could potentially be alleviated by splitting up the code into distinct executables, or docker containers. However, there are a couple of drawbacks with this strategy as well. To begin, an inter-process communication (IPC) mechanism would be required to convey, e.g., the smart contract outputs. Second, we would either need a new executable (or container) for each interface exposed by FEVER's smart contracts, or we would have to supply command-line options that complicate invocation. Third, this approach is in stark contradiction with the microservice paradigm, which FEVER's architecture employs. Fourth, and perhaps most significantly, it ties up the implementation of FEVER's off-chain components with a specific blockchain technology, i.e., HLF in this case. Arguably, it would be preferable to completely abstract the employed technology in order to deliver a more modular and portable solution.

The second, and preferable, method is to split the code that interacts with the blockchain network into a distinct microservice, with the goal of making it as blockchain agnostic as feasible. A further need is that this microservice's core codebase is as separate as possible from the smart contracts that are instantiated on the ledger. Put simply, if we were to develop and deploy a new smart contract for FEVER, we would prefer not to have to write any additional code to interact with it, at least at the backend level. Lastly, the issue of security cannot be overstated. Indeed, this microservice needs to support industry standard protocols for authentication and authorization (SF 1.3), which will also smoothen the integration with other off-chain components (e.g., UIs and web services). In the following, we provide an overview of the HLF smart contract gateway, a microservice that tackles all of the aforementioned challenges.

We begin our description by presenting, in Figure 9, a high-level overview of the APIs exposed by the HLF smart contract gateway. As is illustrated in the figure, the first set of APIs that the gateway exposes provide a direct mapping between the functions that are exposed by the instantiated smart contracts on the ledger. Furthermore, this component exposes a credential management API that allows an administrator of the EC operator to insert, update and revoke the private key and the corresponding digital certificate of, e.g., an EC peer. These credentials are employed by the gateway for transaction signing purposes when an authenticated user invokes the API corresponding to an exposed smart contract function. The interaction of the gateway with FEVER's blockchain network and the contracts that are instantiated therein is based on the gRPC protocol.

The HLF smart contract gateway's configuration is comprised by two parts, which we introduce in the following. The network connection profile is the initial file, and it contains information about the target blockchain network that is required for the application to interface with it. This is a YAML file that contains the following sectioned data:

- The orderers and peers that the application has specified to use for transactions for each channel, as well as a set of optional policies that the program can use to accomplish channel operations.
- A description of the network's participants, which includes information such as their MSP's identity and the links to their MSP's root certificates.
- The orderers to whom the service can send transactions and channel create/update requests.
- Connection information for the peers to which the gateway can submit various requests, such as transaction endorsements, queries, and event listener registrations.

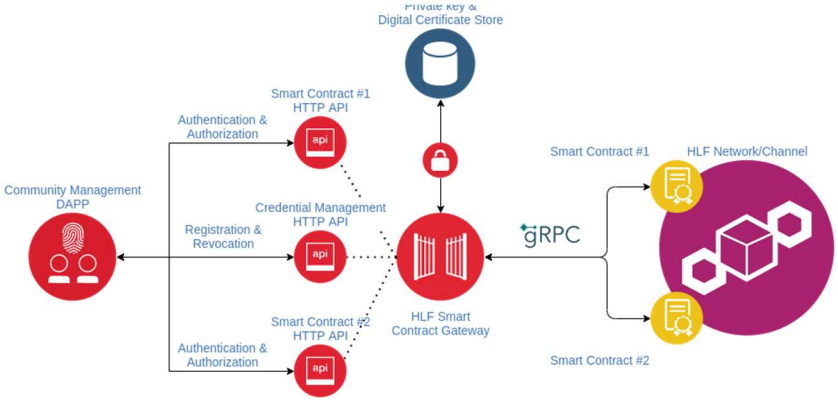

Figure 9: Overview of the APIs exposed by the HLF smart contract gateway, its interactions with the community management DAPP and the smart contract abstractions it provides.

The gateway's second configuration file contains information about the HTTP endpoints it will expose and the smart contracts it will interface with. The path to the network connection profile comes first. Second, the HTTP listening port of the gateway. Third, and most significantly, a section dedicated to giving all of the information required by the gateway in order for it to not only communicate with the smart contracts, but also expose their functions as a REST API. This part specifies the smart contract's instantiation name and the channel on which it is instantiated. This configuration file also includes the following information for each smart contract function:

- The function's name, as it is exposed by the contract.
- The HTTP method name that will be employed, i.e., GET for queries and POST for transactions.
- The name of the endpoint that is appended to the base URL of the gateway to invoke this function.
- An optional list of URL parameters, based on the REST paradigm, which can be encoded.
- A specification of response headers.

In the context of POST methods, the body of the request is presumed to be in JSON format at all times. The URLs that the gateway exposes for each smart contract function have the following format:

```
https://<ip>:<port>/<channel>/<contract>/<function_endpoint>[/<URL_parameter>]
```

5.6 On-Boarding EC Members

In this section, we provide an intuitive description of the process that we envision for on-boarding on FEVER's blockchain-based ecosystem of services a new EC peer, to which we will refer as Carol. An overview of the steps involved in this process is illustrated in Figure 10.

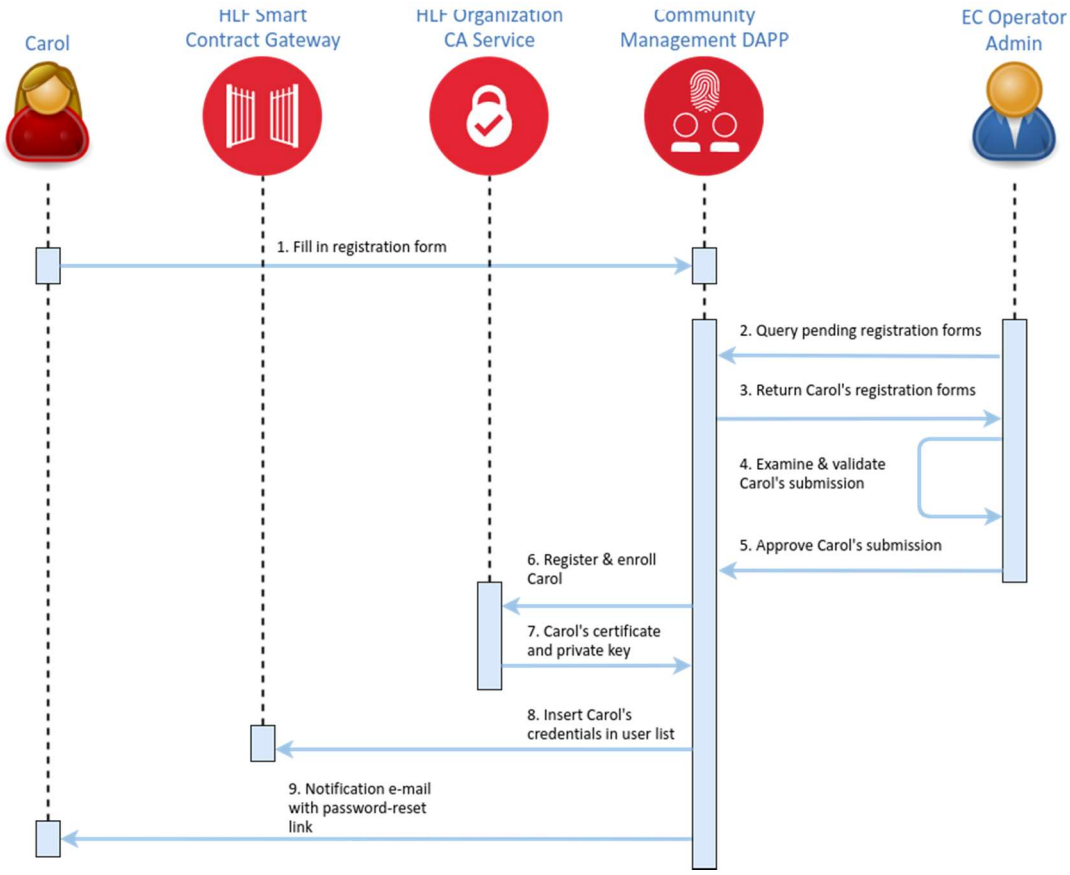

Figure 10: Sequence diagram depicting the interactions involved to successfully on-board a new EC member in the community management DAPP ecosystem.

Carol navigates to the website of her respective EC operator and clicks on the registration button. This will redirect her to the community management DAPP that displays a registration form. Common fields that we envision in this form are name, surname and an e-mail address. However, depending on the specific deployment, it could be the case that Carol would be required to upload additional real-life documents pertaining to, e.g., the assets that are available to her premises. Carol submits the form (Step 1), which is registered and marked as pending in the internal state of the community management DAPP. The UI displays her a message that she will be notified with an e-mail message by the EC operator's personnel following the review of her request. As an alternative, and depending on the real-world deployment scenario, this first step could be performed by an administrator of the EC operator who has privileged access to the community management DAPP ecosystem.

Regardless of the first step's concrete realization, at some point in time, an administrator of the EC operator will log on to the web UI of the community management DAPP where she is presented with a management dashboard that she can employ to, on the one hand, view and manage the credentials and relevant PII associated with already registered EC members, as well as, pending registration forms. The administrator of the EC operator examines and validates the information provided by Carol, along with any other (real-world) documentation that might be supplied as attachments and, subsequently, approves her registration by clicking the corresponding button in the web UI. In the backend, this will trigger the following steps. First, the registration and enrollment of Carol to the EC operator's CA service, which will output Carol's private key and corresponding X.509 public key certificate. Second, these credentials will be added to the smart contract gateway's user list by invoking the appropriate endpoint this service, access to which is restricted to the EC operator's administrator. Once these steps are completed, the community management DAPP will send an e-mail message to Carol that includes a link that will allow her to specify her login password. Once Carol does so, she can employ her e-mail address as her username, along with the password of her choice, to interact with FEVER's platform.

5.7 Retracting EC Members

In this section, we provide an intuitive description of the exact opposite process that was presented in the previous, which revolves around retracting membership of an EC peer from FEVER's blockchain-based ecosystem of services. Figure 11 provides an overview of the involved steps in this process.

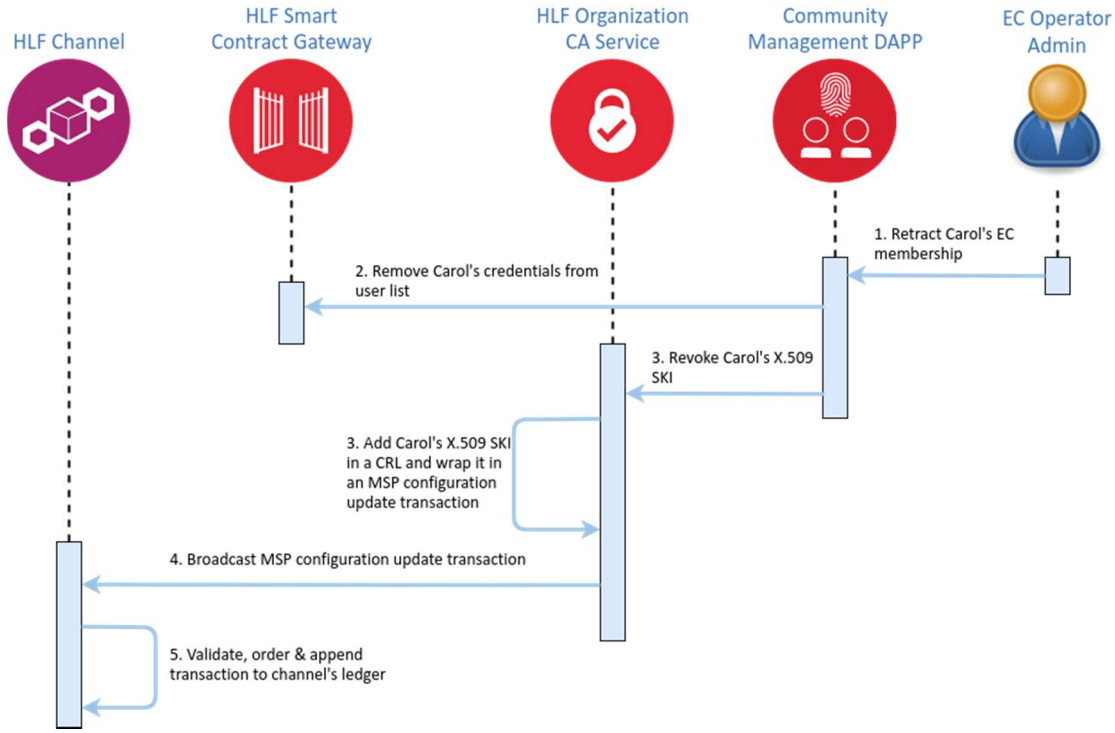

Figure 11: Sequence diagram depicting the interactions involved to successfully retract an EC member from the community management DAPP ecosystem.

In the interest of providing a simplified description, we assume that this process is initiated by an administrator of the EC operator. As was the case in the previous section, the administrator logs in to the web UI of the community management DAPP, which displays a list of the admitted members. By employing the search function of the UI, the administrator can quickly locate the record that corresponds to Carol, which we assume is the member to be retracted. By clicking on the record pertaining to her registration, the administrator is presented with a card view displaying additional information about Carol. The card includes a red colored button entitled "Remove" that, on click, will trigger the following process (a warning message about what is going to take place might be displayed as a countermeasure for accidental clicks). The first step that occurs at the backend is removal of Carol's credentials from the HLF smart contract gateway, which will effectively block her ability to issue transactions to the ledger. Second, the secure revocation process involves the invocation of the CA service's appropriate endpoint which, on input a user ID, generates a CRL that includes the user's corresponding SKI, wraps it an envelope-contained MSP configuration update transaction and broadcasts it to the network. Once the ordering service has successfully validated and appended the aforementioned transaction to a newly formed block that is propagated to the HLF peers of the channel, Carol's credentials have been successfully revoked from the community management DAPP ecosystem.

6 FlexCoin DAPP Specification

6.1 Background

One of the focal objectives of the FEVER project is the design and implementation of a secure, decentralized and verifiable peer-to-peer (P2P) energy/flexibility trading framework that will be inclusive for energy communities (ECs). The inherent decentralization of blockchains and, more specifically, smart contract platforms, which are considered as the second generation of blockchains, provide the technical means to address these security requirements via their append-only, verifiable and fault-tolerant transaction log.

At the core of any trading system, regardless of its application domain, lies, what is commonly referred to, a *common value system*, i.e., an exchangeable unit, or currency, which, on the one hand, has value for all participants in the trading system and, on the other hand, is used as a building block to trade products and services at various marketplaces that are built on top of it. In the context of distributed ledgers, blockchains and smart contract platforms, currencies of this type, i.e., whose units are indistinguishable from one another and, as a result, represent the same value, regardless of its concrete quantification, are, on a high-level, categorized as follows:

- **Cryptocurrencies.** In public blockchains, whose underlying consensus protocol typically revolves around proof-of-work (PoW), or even proof-of-stake (PoS) in a few cases, the term *cryptocurrency* is employed. However, since in FEVER we employ Hyperledger Fabric (HLF), which is a permissioned smart contract platform, it is evident that we are in need of another token mechanism.
- **Fungible Tokens**. The second category of the tokens discussed here is commonly referred to as *fungible tokens* (FTs), the most prominent example of which is the ERC-20 [11] token standard. These are typically implemented via a smart contract and follow an account, or bank-like model that we are all accustomed with. Put simply, each user has one or more accounts in the system (smart contract), each with its own separate coin balance that users can trade amongst them. For the sake of simplicity, we assume that no restrictions in regard to value transfer apply in our case, contrary to the heavily regulated framework of the real-world banking system.

Although the ERC-20 token standard serves as the de-facto reference point for systems that require an FT as a building block for arbitrary trading services, it has several drawbacks:

1. Our first line of criticism stems directly from the fact that the ERC-20 token standard was developed by having solely public smart contract platforms in mind, i.e., Ethereum in this case. Indeed, ERC-20 compliant smart contracts emit an event when, e.g., tokens are transferred between accounts, which in the context of public blockchains is considered acceptable, since by definition they do not provide for privacy. However, in contexts where privacy matters, as is the case for FEVER, such mechanisms need to be eliminated. We stress that the ERC-20 standard involves several other events that, ultimately, violate even further the privacy of end-users. On a similar note, the event mechanism of blockchains has to be used in moderation, as it imposes a significant performance penalty and computational burden on the peers of the network. Put simply, it should not be treated as a generic notification mechanism, i.e., along the lines of push notifications on mobile environments.
2. In regard to the functionalities provided by the ERC-20 token standard; the following points are warranted. The standard is arguably riddled by superfluous functions that, even in public blockchains, are never used. The most prominent example relates to the functionality of its Approval API which, in short, allows account owners to approve the transfer of specific amounts of funds from their account by other designated users. Based on our experience from a variety of other H2020 projects in various applications domains, e.g., energy, health, transportation and others, end-users have reported either disinterest, or even privacy concerns and general discomfort in some fewer cases, regarding this functionality.
3. The ERC-20 token standard, as well as, several other proposals built on top of it, lack a concrete mechanism for creating and removing value from the system, i.e., processes for minting and burning coins, respectively. For instance, the standard is based on the assumption that the number of coins, along with all their available denominations, are specified during the contract's

instantiation on the blockchain. That being said, we are now in the position of highlighting a subtle, yet relevant issue in the context of FEVER that is related to this discussion. On a high-level, membership in an EC and, by extension, to the overall P2P trading system that is developed in the project, should not be treated as static. Put simply, participants are expected to come and go. Hence, for departing parties, it is evident that the value that they have accrued in the token system has to be, in some cases, translated, or mapped, to some agreed-upon value in the real-world. This alone showcases the necessity for supporting a process for burning coins. Similarly, newly arriving participants in the system, by definition, will not have any value, or balance, in the token system, i.e., they will be unable to participate in the trading services that are built on top, which in turn highlights the need for a process to mint coins. Clearly, in order to provide for sensible coin minting and burning mechanisms, it is imperative to design a binding mechanism among the transactions that take place on the ledger and their effects, or meaning, in the real-world. We stress that, from a security perspective, this involves the resolution of the following challenges.

a. With regards to privacy, it is preferrable that the binding mechanism does not leek any information in regards to, e.g., the number of coins that are minted, or burned, as well as, the identity of the account owner (based on our definition of privacy).
b. The mechanism itself has to be secure, i.e., it should not allow malicious parties, or behaviours thereof, to go by undetected. From a cryptographic point of view, we stress that in 2-party protocols, if one of the involved parties is malicious, then the best possible outcome is for the other (non-malicious) party to abort the protocol, along with (cryptographic) proof of the other party's deviating behaviour. For instance, in the context of FEVER, and based on the deliverable's description up to this point, it is envisioned that the minting and burning processes will take place between the EC operator and, e.g., an EC member. In this setting, we must ensure that, regardless of which of these two parties deviates from the protocol, there will be readily available proof on the ledger that can, e.g., be supplied to some external auditor to resolve the dispute.
c. It would be desirable if the binding mechanism were generic, i.e., the protocol's involved commitments were independent of the respective real-world processes and potential documents involved. In the overwhelming majority of modern trading paradigms, whether they are digital or not, there comes a point in the trading process where the buyer commits to her choice of purchasing a particular good or service. For instance, in auctions, once a bit is submitted, the bidder is bound, or expected, to uphold her end and pay the incurred price, assuming of course that she is not outbid. Put simply, defrauds, or any other instance of deserting behaviour is not permitted, or even tolerated.

In FEVER, to provide for a secure trading framework that will, in turn, promote trust and future adoption, it is required by the underlying token system to provide, loosely speaking, a "digital equivalent" of the aforementioned guarantees. We stress that, to the author's knowledge, none of the proposed FT systems does so in a secure, privacy-preserving, self-sovereign and decentralized fashion.

6.2 FlexCoin Design

In this chapter, we present the design of FlexCoin, a secure, verifiable and privacy-preserving smart contract ecosystem for modelling FTs that can also be used as a stable coin, i.e., a digital currency that is pegged to a real-world fiat. At the core of our construction lies a novel mechanism for notarizing arbitrary operations (functions) exposed by smart contracts which, in short, separates the code of the smart contract's execution, i.e., the implementation of the business logic, from the policies that govern its execution, i.e., access control and authorization policies that determine, in real-time, the conditions under which the operation should be executed (or not). This notarization mechanism promotes, among others, code modularity and, due to it being completely agnostic, by design, to the smart contract operation and the policies governing its execution (if any), it is generic and expressive enough to be applicable across a wide range of use cases and is, thus, of independent research interest. In terms of exposed functionality, FlexCoin provides substantial improvements to the ERC-20 standard, which can be summarised as follows:

- It eliminates superfluous functionalities that, in the author's experience, have no practical use and, due to it being tailored for permissioned environments, provides for increased privacy and scalability.
- FlexCoin provides secure protocol implementations for minting and burning coins that is suitable for real-world applications. Thus, it resolves one of the main limitations of the ERC-20 token standard, which requires the specification of the total number of coins and its involved denominations (if any) during setup, i.e., when the smart contract is instantiated on the ledger.
- To facilitate the design and development of arbitrary digital marketplaces that involve financial commitments among transacting parties, similar to the ones that are applicable in real-world trading scenarios, FlexCoin provides a novel "coin hold" mechanism. In a few words, this mechanism locks a portion of a user's account balance, thus, rendering it completely unusable. Subsequently, the marketplace smart contract, based on its respective decision-making process, to which FlexCoin is completely agnostic, determines whether the held coins should be returned to the user's active balance, or if they should be (even partially) transferred to one, or more, accounts of other users.

The basis for FlexCoin's design and implementation stems from the authors' efforts in the RENAISSANCE H2020 project which, coincidentally, also revolves around ECs and their integration in various digitized energy and flexibility marketplaces. In the interest of clarity, we stress that in the context of FEVER, the FT framework has undergone severe (technical) modifications and extensions, which are summarised as follows:

- Separation of the functionality provided by the FT notary smart contract from that of the EC's governance to provide for increased isolation and modularity, among others.
- Simplification of the FT notary interface in the context of mint and burn request notarization, both in the centralized and the decentralized setting. We stress that in the context of FEVER, we are interested solely on the centralized version, however, the changes are transparently inherited to the decentralized version as well.
- Extended the centralized FT notary smart contract to support multiple financial principals (or governance body members as they were referred to in the context of RENAISSANCE) for increased service availability.
- Extended the FT notary smart contract to support notarization of access control policies for privileged functions that should be restricted to specific actors, such as querying historical mint requests, which is limited to the EC operator in the context of FEVER.
- Extended the FT smart contract to allow for multiple accounts per user. This involves a radical restructure of the contract's internal state organization, among others.
- Modified the account ownership model to be based on the combination of the user's common name attribute and the identifier of her respective MSP, instead of a public key. In short, this provides for immense flexibility in regards to digital certificate lifecycle management, facilitates and promotes storage of digital credentials across multiple edge devices controlled by the user.
- Along the lines of allowing for multiple accounts per user, we have extended the FT smart contract to allow for multiple concurrent mint and burn requests on a per account basis. This is a substantial improvement to the previous version, which only allowed one active mint and burn request per unique user account and, as is the case with adding support for multiple accounts per user, requires a complete overhaul of the contract's internal state organization.
- Extended the API of the FT smart contract to provide a structured, verifiable and totally ordered log of the actions performed on each individual account.
- Extended and slightly simplified the API of the FT smart contract by adding a set of privileged functions that will allow financial principals (e.g., the EC operator) to manage mint and burn requests more efficiently.

6.3 Smart Contract Notarization

In this section, we progressively introduce our novel mechanism for notarizing arbitrary operations (functions) that are exposed by smart contracts, to which we will refer to from hereon in as the notary interaction pattern. We begin by providing an abstracted description that introduces fundamental

concepts and the involved entities in this paradigm. To facilitate reader understanding, we subsequently present concrete use cases of this pattern in the context of FEVER's FlexCoin.

The entities that are involved in the notary interaction, along with a short description of their role, is as follows:

1. Service Contract: An arbitrary smart contract that exposes one or more operations (functions) that can be invoked by clients. This is where the business logic of the operation is implemented.
2. Client: An entity (human or not) that wishes to invoke an operation of the service contract.
3. Notary Contract: A smart contract that governs the policies under which operations invoked by clients are executed or not. This is where access control and authorization, among other policies, are implemented.

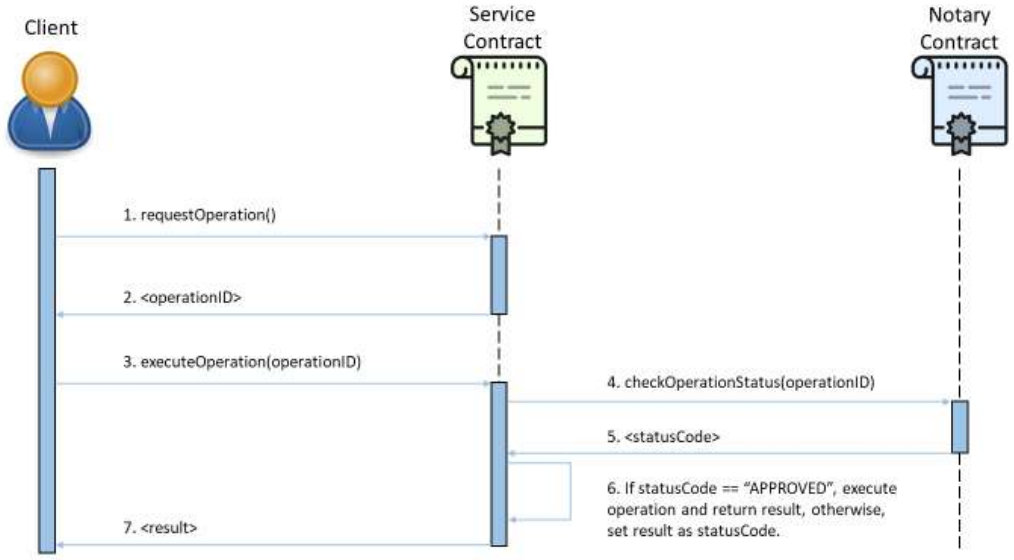

Figure 12: Sequence diagram illustrating the interactions among the entities involved in the notary interaction pattern.

In Figure 12, we provide an abstract illustration of the interactions among the entities (enlisted above) that are involved in our notarization mechanism. In the depicted example, we assume that the service contract exposes, for the sake of simplicity, a single notarized operation. As it will gradually become apparent, the functionality of the operation at hand is completely irrelevant in the pattern's description.

The process is initiated by the client issuing a request to the service contract's notarized operation (Step 1), which is illustrated as the invocation of the function "requestOperation()". The service contract, presumably performs some preliminary internal processing and, ultimately, assigns and returns to the client a unique identifier for her request (Step 2). We stress that this step does not involve the execution of code that is part of the operation's business logic. At some future time, e.g., when the conditions that will permit the **actual** execution of the client's request are met, the client invokes the service contract to execute the operation by supplying as input the identifier of her operation's instance (Step 3), which is illustrated as the invocation of the function "executeOperation()". In turn, the service contract will invoke the notary contract for this particular operation (Step 4) and will receive, as a response, a status code regarding this instance of the operation (Step 5). The service contract uses this status code to infer whether it should proceed (or not) to the execution of this operation's instance (Step 6). Lastly, the service contract supplies the client with the corresponding result (Step 7).

To provide further insight in regard to the notary interaction pattern, we highlight the following points. First, a successful completion of the pattern requires a total of two transactions, assuming a happy-path scenario, i.e., Steps 1-2 occur in the context of the first transaction and Steps 3-7 in the context of the second transaction. Second, the pattern is asynchronous by design, i.e., there are no inherent timing

assumptions regarding, e.g., the time window between the pattern's transactions. Third, there is a direct association (mapping) between each notarized operation of a service contract and its respective notary or, more specifically, the operation's notarization function. A service contract can employ a potentially distinct notary contract for each of its notarized operations. Thus, on the one hand, the implementation of a notary can be shared across multiple service contracts and, on the other hand, notaries can be dynamically updated, i.e., either from the service contract's perspective, or even regarding the implementation of the notary itself. Fourth, any peculiarities regarding the implementation of the service and notary contracts are completely abstracted from the protocol. For instance, both contracts can have their own, e.g., access control policies, or may even be controlled under centralized or decentralized schemes. Lastly, on a more implementation-related note, notice that from the point of view of the service contract, the notary is simply a function call. In HLF, this requires that the service contract is configured with the following values: 1) the name of the channel on which the notary contract is instantiated, 2) the instantiation name of the notary contract and, 3) the name of the notarization function for a specific operation. Consequently, it is imperative that the notary contract is instantiated on the ledger prior the service contract and that both contracts are in the same channel, since cross-channel contract invocations do not allow for modifications (writes) on contract state.

In the following, we provide a descriptive list of the status codes that a notary contract can return as responses to its notarization function:

- **NOT_FOUND:** The notary is unaware of an operation instance corresponding to the input identifier.
- **PENDING:** The notary is aware of the operation's instance, however, an affidavit regarding its execution is not, as of yet, available.
- **APPROVED:** The policies governing the execution of this operation's instance have been met and the service contract can proceed in executing the involved business logic.
- **REJECTED:** Following the evaluation of the policies governing the execution of this operation's instance, the service contract is instructed to not proceed with the execution of the business logic.

The aforementioned status codes have been found to be sufficient for a wide range of use cases. However, designers and developers are free to extend the set of supported status codes to fit the peculiarities of their use case. Moreover, we stress that the "contract" conveyed by these status codes, i.e., their particular meaning is use case specific. For instance, on receipt of a "REJECTED" status code, the service contract could some other business logic prior to responding to the client.

6.3.1 Use Cases

ECs are comprised by a diverse set of participants, ranging from typical end-users, such as consumers and prosumers, to a variety of energy stakeholders and public organizations, all of which cooperate in the generation, distribution, storage and balance of the underlying grid. On a high-level, a focal objective of FEVER is to facilitate secure and verifiable peer-to-peer (P2P) energy trading that can provide for real-world financial compensations. To provide for settlements that have this property, there need to be clearly defined processes that address the following issues: 1) creation of value, or minting of coins that will be used as a basis for trading and, 2) burning of coins, which is the exact opposite of minting and, in short, allows participants to "cash-out" accrued earnings from trading in the system into real-world fiat. Clearly, in the decentralized setting put forth by ECs, orchestration of these processes has to provide for strong security and verifiability to be deemed trustworthy, as per the requirements of Section 3.

To facilitate reader understanding, we begin by citing, on a high-level, a set of challenges involved in practically realizing the conversion of monetary value between the real and the blockchain world, which will also shed light to the rationale based on which they were addressed in FlexCoin. Arguably, any real-world organization has a principal that oversees its financial activities, ranging from, e.g., a single accountant, to more complicated structures, such as entire financial departments. In addition, the governance of this principal can be centralized, i.e., under the control of a single management structure, or even decentralized, e.g., where multiple, distinct administrative domains participate and influence decision making. Although in FEVER the focus is steered towards centrally-governed financial principals

for ECs, where the involved responsibilities are undertaken by the EC operator, it would, nevertheless, be preferrable to provide for a technical design that allows the minting and burning of coins to be realized, if need be, in a decentralized fashion. Irrespective of the nature of the financial principal involved in these processes, it is of vital importance to impose boundaries in terms of its control over them, which is of particular relevance and interest to guarantee a balance of power, especially when dealing with end-users. For instance, it should **not** be the case that the principal could directly affect the coin balances of accounts of end-users, such as LEC members, e.g., by subtracting sums of coins without the explicit consent of the account owner.

We dedicate the following subsections on providing concrete examples that showcase the ability of the notary interaction pattern to resolve all of the aforementioned challenges in a practical, real-world setting. More specifically, we describe the means under which coins can be minted (Section 6.3.1.1) and burned (Section 6.3.1.2) in the FlexCoin smart contract ecosystem, which is comprised by the following two smart contracts. First, the FlexCoin FT smart contract, which provides the implementation of the FT functionalities (Section 6.5). Second, the FlexCoin governance smart contract, to which we interchangeably refer to also as the FlexCoin notary smart contract, which acts as the notary of the FlexCoin FT smart contract (Section 6.4). As previously stated, we assume a centrally-governed financial principal, a role which is concretely undertaken by the EC operator. However, in Section 6.3.1.4, we provide a short, high-level description on how these processes can be decentralized via the notary interaction pattern. Lastly, in Section 6.3.1.5, we discuss a more elaborate example that is related with a separate functionality supported by FlexCoin, i.e., the coin hold mechanism. In short, this is a proposal for providing the digital equivalent of financial commitments that are typically involved in real-world marketplaces and is also based on the notary interaction pattern.

6.3.1.1 Minting Coins

Assume Carol, a member of an EC that wishes to join FEVER's blockchain-based energy/flexibility trading ecosystem. Carol will have to first register to her EC's web portal, which is provided as a service by the EC operator. The details of the registration process are not relevant in the context of this discussion (refer to Section 5.6 for more details). Following this step, Carol logs in with her newly created account and navigates to the FlexCoin DAPP tab. As she has no FlexCoin account(s), she will be prompted with appropriate instructions by the UI to create one. Carol's newly created FlexCoin account has a zero-coin balance, which limits her ability of engaging in energy trading, i.e., she is unable to participate as a buyer on the available marketplaces as that requires a positive FlexCoin balance. However, she can request the minting of coins by clicking on a button next to the entry of her newly created account. A pop-up appears on Carol's screen prompting her to input her credit/debit card details, similar to the case of how standard digital payments are carried-out on the web. Assuming the payment is carried out gracefully, Carol is, subsequently, prompted by the EC operator's web platform to upload a copy of her bank's digital receipt (e.g., in PDF format), which serves as a claim, or commitment, that she has performed the payment. In the background, a verifiable, privacy-preserving representation of Carol's bank receipt is computed, which is posted on the ledger along with, e.g., the requested amount of FlexCoins to be minted. We discuss later on in this section available options on computing verifiable representations of data. Carol's receipt can be stored in, e.g., the portal's database. Following the completion of these steps, Carol's FlexCoin DAPP view is updated to display her newly issued mint request, whose status is set on pending.

From the point of view of the EC operator, Carol's newly issued mint request is presented as a new entry to the management tab of the operator's FlexCoin DAPP view. For each mint request, the operator is able to examine the amount of FlexCoins requested to be minted, as well as, the attached bank receipt. Conceptually, the operator is expected to verify the validity of Carol's request which, on a high-level, involves the following: 1) examining the attached bank receipt to infer its authenticity, i.e., that it was issued by a trusted bank (this could be strengthened in the future when digital signatures are, by default, attached to documents to verify their authenticity), 2) checking that the requested amount of FlexCoins to be minted equals (recall we assume a one-to-one conversion rate for simplicity) the sum paid by Carol and, 3) the operator could, potentially, check her own bank account to further confirm that the payment has been processed correctly. We stress that the digital receipt attached to a mint request, prior to it being visualized to the operator, is verified against the value that was posted on the ledger as

part of Carol's mint request. The immutability and tamper-evidence properties of the ledger, combined with the verifiability property of Carol's computed value ensure, in short, that even if, e.g., the database where the receipt is physically stored becomes corrupted by some malicious party, it will be detected and, subsequently, appropriate corrective actions can take place. Naturally, all of the aforementioned verifications, provided the appropriate technical tools and infrastructure, can be automated entirely. Following the successful validation of Carol's request, the EC operator approves it via the UI of the FlexCoin DAPP management tab. On the backend side, this leads to the issuance of two transactions, an overview of which is as follows. The purpose of the first transaction is to approve the execution of Carol's mint request and the second executes it on her behalf, which results in Carol's account being credited with the designated amount of FlexCoins. In summary, this process allows us to address the requirement SF 1.2.2.

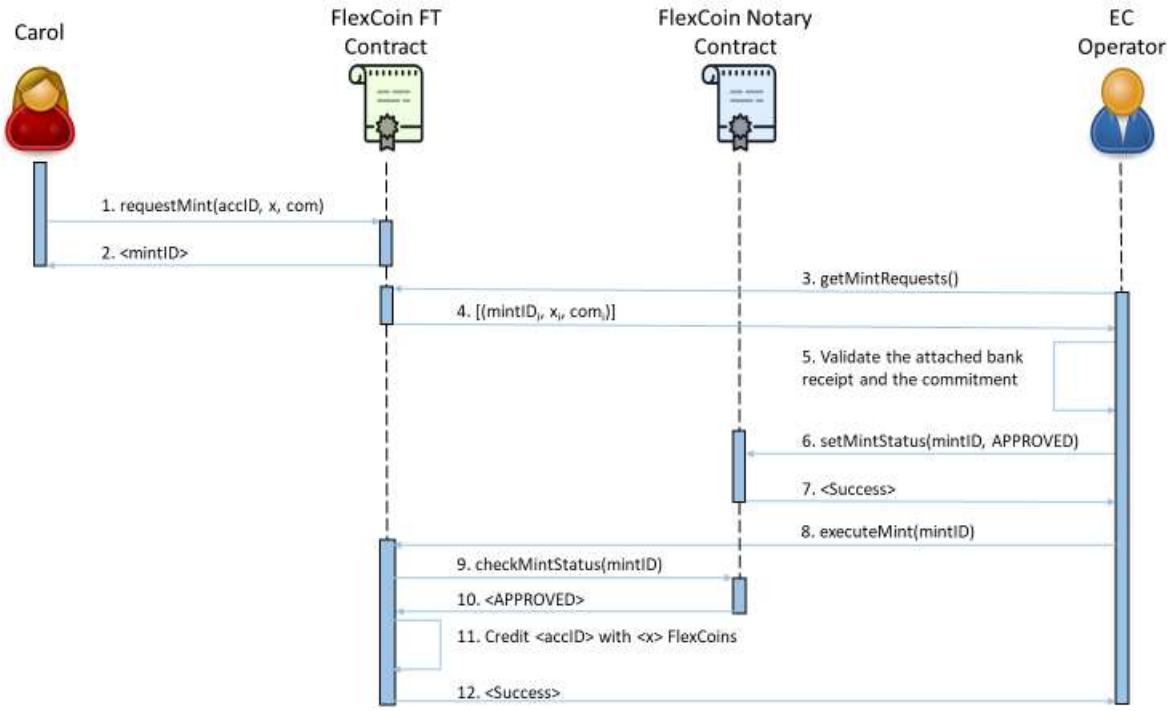

Figure 13: Sequence diagram depicting the interactions involved to successfully complete a mint request in the FlexCoin smart contract ecosystem.

Following the conceptual description provided above, we are now in the position of elaborating further regarding the internals of the minting process. In Figure 13, we provide a sequence diagram depicting the interactions among Carol, the party requesting the minting of FlexCoins, the FlexCoin smart contract ecosystem, which is comprised, as illustrated and discussed in the previous section, by two separate smart contracts, i.e., the FlexCoin FT and the FlexCoin notary contracts, and the EC operator, whose main role is to validate and approve/reject the request. Notice that the involved steps correspond to the generalized ones that were presented as part of the description of the notary interaction pattern. Moreover, in the illustration of Figure 13, the steps involved in the notary's decision-making process for this particular use case are also provided, which were assumed out as part of the generic description of the notary interaction pattern.

In the remainder of this section, we discuss subtle details related to particular steps of the presented minting process. We begin with the computation of the value that provides for verifiability of Carol's digital receipt (Step 1), i.e., her claim, or commitment. We briefly discuss involved trade-offs between security, privacy, trust and performance among possible alternatives, which are the following:

1. **Identifiers:** The first and simplest to implement alternative involves the assignment of an identifier that "points", either directly, or indirectly, to, e.g., the database in which the receipt is physically stored. This does not provide for any level of security and verifiability, as its correctness is predicated on the assumption that the database is completely non-malleable. Depending on the specifics regarding the generation of this identifier, or hash link as it is

commonly referred to in such contexts, it can provide for an acceptable level of privacy and is, arguably, very efficient, in terms of overall performance-related metrics.

2. **Cryptographic Hashes:** In this alternative, the value posted on the ledger is a cryptographic hash of the receipt, which is computed by employing a collision-resistant hash function, e.g., SHA256. This approach provides an attractive combination of properties in terms of all considered axes. Indeed, computing a single hash is efficient and simple to implement in software. Moreover, it eliminates trust on the external database that is employed for storage and provides a strong level of security and, most importantly, verifiability. Regarding privacy, strictly speaking from a formal cryptographic point of view, hash functions do not provide hiding guarantees. However, in practical terms, their use is encountered in numerous real-world, production-grade deployments to, falsely, provide for privacy. The main trade-off regarding this matter is, simply put, how much information is available to the adversary and the level of randomness involved (if any) in the input payloads.
3. **Cryptographic Commitments:** This constitutes, arguably, a broad category that can be used as an umbrella term to encompass, on the one hand, multiple concrete technical constructs based on different security assumptions and, on the other hand, their amenability to be combined with advanced privacy-preserving techniques stemming from the field of zero-knowledge. In a few words, this approach has the potential of providing the highest level of guarantees regarding security, privacy and verifiability in the expense of, however, implementation complexity, i.e., it is not developer-friendly. In terms of computational complexity, i.e., performance, although these techniques are far more heavy-weight than the previous alternatives, given the fact that mint requests are projected to be fairly infrequent and, in addition, are validated based on human intervention, the involved latency of this process will mask the incurred performance hit.

The FlexCoin smart contract ecosystem is agnostic to the employed construction for computing and verifying claims. We stress that this is a conscious design choice as it abstracts the intricacies of the employed claims scheme, which allows these smart contracts to be re-usable. Moreover, since the validation process, as was previously described, from a business-logic perspective, takes place off-chain, the smart contract's active involvement, i.e., from a computation point of view, is not required. However, the smart contract's state is employed to ensure durability of claims, which serves as the main anchor for the verifiability, auditability and, ultimately, accountability properties of the presented minting process.

The final part of the minting process, which involves the invocation of the FlexCoin FT smart contract to execute an approved mint request, as is depicted in steps 9-13 of Figure 13, does not, necessarily, depend on the EC operator. Indeed, depending on the specifics of the real-world scenario, these steps could, alternatively, be executed by the entity that issued the mint request (Carol). We stress that this restriction can be enforced programmatically by appropriately tuning the policies of the notary contract. However, we believe that the depicted approach provides for improved user experience, as it does not require further involvement by the requesting entity, following the completion of the notary's decision-making process.

6.3.1.2 Burning Coins

The complement of the minting process is commonly referred to as burning in the fungible token literature. Put simply, this is a mechanism that allows an entity to "cash-out" of the system. We build on top of the same paradigm that was employed in the previous section to facilitate reader understanding. Assume that Carol has accrued a positive balance of FlexCoins by, e.g., participating in the various energy-related marketplaces, and that she wishes to exchange them with real-world currency. Carol logs in to the LEC's web portal and navigates to the FlexCoin DAPP tab where she is presented with an overview of her FlexCoin accounts and their respective balances, among others. For each account entry, next to the button that allows her to initiate the minting process, lies a button that will allow her to burn coins. On click, Carol is presented with a pop-up that prompts her to select one of her FlexCoin accounts, the amount of FlexCoins that she wishes to exchange and, lastly, the IBAN of her real-world bank account to which the exchanged (again assuming a one-to-one correspondence for simplicity) fiat currency units will be deposited. Following the submission of this simple form, Carol's FlexCoin DAPP tab is updated to display her newly issues burn request, whose status is set to pending. We stress that

these coins are “locked”, i.e., they are not part of the account’s active balance and are, effectively, rendered unusable until the burning process concludes (successfully or not). In summary, this process allows us to address requirements SF 1.3 and 1.7.

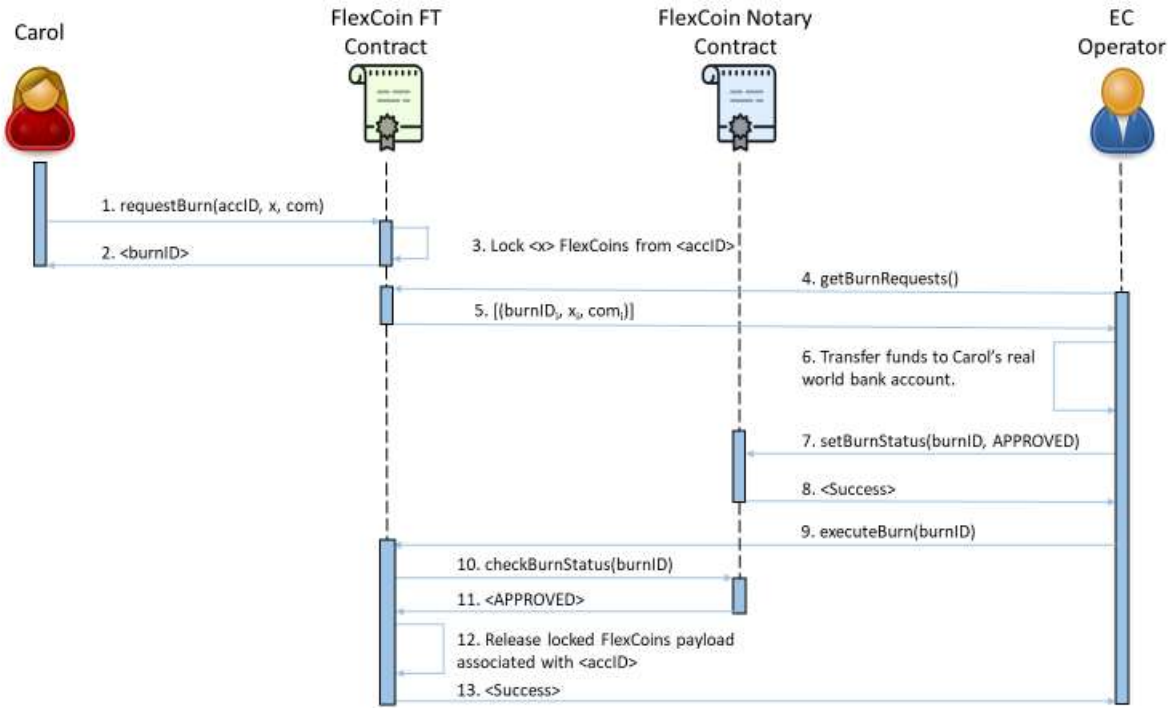

Figure 14: Sequence diagram depicting the interactions involved to successfully complete a burn request in the FlexCoin smart contract ecosystem.

Switching to the perspective of the EC operator, Carol’s newly issued burn request is visualized in the management tab of the operator’s FlexCoin DAPP view, as is the case with mint requests. In a similar fashion, for each burn request, the operator is able to examine the amount of FlexCoins to be exchanged, as well as, the IBAN to which units of fiat currency should be deposited. The EC operator is now expected to complete the wire transfer to Carol’s real-world bank account, a statement which warrants further clarification. Strictly from a technical point of view, the EC operator is, clearly, not forced to do so. However, we assume that appropriate contractual agreements are in place which, potentially, could take place during the on-boarding/registration to the system, e.g., a service-level agreement (SLA), which is commonplace across transactional, real-world digital services. In the paradigm discussed here, it is reasonable to assume that a subset of the agreement’s terms imposes explicit temporal restrictions for various processes. For instance, the EC operator, following the issuance of a burn request, could be obligated to complete the wire transfer in the span of, e.g., three business days. Since Carol’s transaction is timestamped on the ledger, it is evident that the blockchain can be employed as a resilient and verifiable source of truth to infer if the conditions of the SLA have been violated (or not). In addition, bank account receipts can serve as complementary sources of proof to infer whether any party has misbehaved, or in general to ensure that the wire transfer has been completed. Following the completion of the wire transfer, the EC operator marks it as approved in the FlexCoin DAPP management tab which, in the background, triggers the following transactions. First, a transaction that marks as approved Carol’s burn request and, second, the execution of the burn request. The latter is a simple clean-up process that releases the burn request data structure that is maintained internally in the state of the FlexCoin FT smart contract.

In the following, we gradually shift from the conceptual description provided above, to more concrete details regarding the burning process. Figure 14 depicts a sequence diagram that illustrates the interactions among the entities that participate in this protocol, i.e., Carol, the FlexCoin FT smart contract ecosystem and the EC. Evidently, the depicted steps resemble the ones of the minting process and, by extension, of the underlying notary interaction pattern.

The initiation step of the coin burn protocol (Step 1) involves a claim, or commitment, from the entity that

issues it. In the previous section, we highlighted several alternative technical constructions that can be employed to implement such schemes in practice. However, contrary to the previous section where we assumed that the nature of the value for which claims are generated (and verified) is relatively fixed, i.e., a bank receipt in PDF format (or another convenient encoding of such a statement), such restrictions do not apply in the context of this discussion. For instance, in the conceptual description of the protocol that was provided above, Carol was prompted to input, among others, her IBAN. This does serve as a viable candidate for the underlying value that is input to the commitment scheme. However, there are several other viable alternatives that can be employed in practice, such as PayPal accounts, credit/debit cards, or even crypto-currency wallet addresses. This flexibility stems from the design choice of abstracting the intricacies of the commitment scheme from the FlexCoin FT smart contract ecosystem, which also encompasses the nature of the value itself, as was discussed in the previous section.

6.3.1.3 Trading Real-World Assets for FlexCoins

In this section, we discuss a simplified example that showcases how the coin minting mechanism provided by the FlexCoin FT ecosystem can be used as a building block to trade real-world assets, such as the ones listed in the requirements section of this deliverable (e.g., bottles of wine, tomatoes etc.), for FlexCoins. For the purpose of the description provided here, we introduce an asset trading smart contract that maintains a mapping between real-world assets and their FlexCoin value.

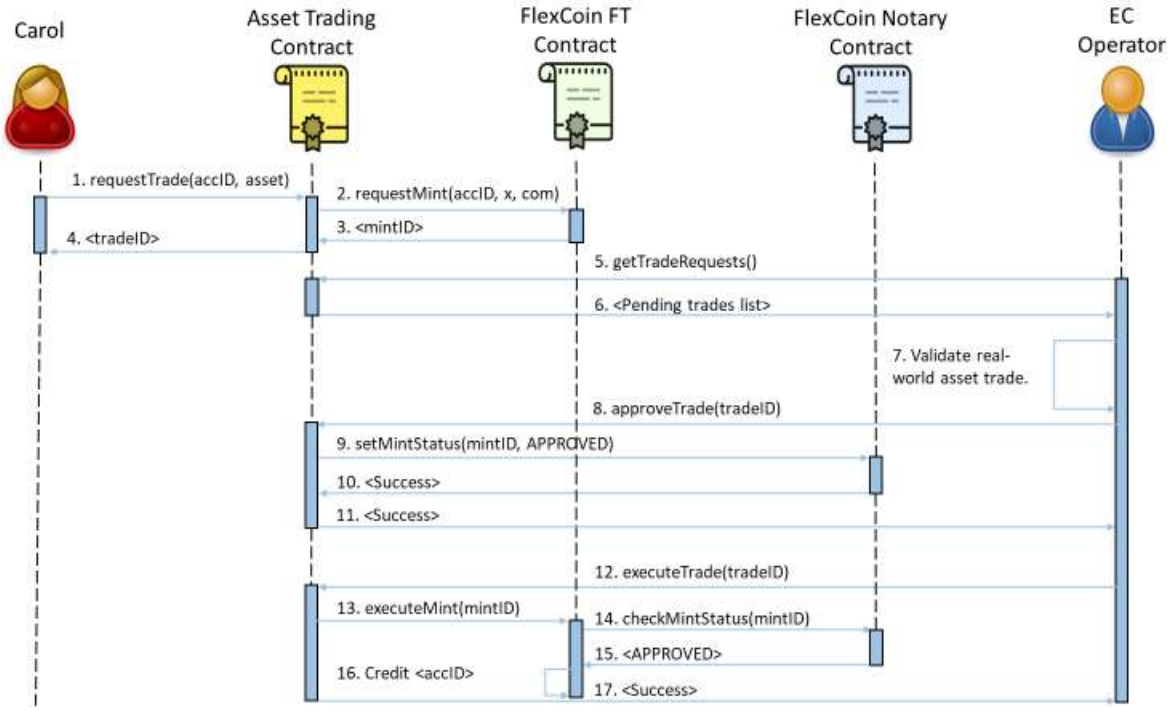

Figure 15: Sequence diagram depicting the interactions involved to successfully trade an arbitrary real-world asset (e.g., bottles of wine) for FlexCoins.

In Figure 15, we illustrate a sequence of steps that can facilitate this type of use case, a high-level description of which is as follows. Carol is assumed to visit some form of facility of the EC where she can physically hand-over the real-world asset that she wishes to exchange for FlexCoins. We assume that there is a separate UI tab that allows Carol to request that her, e.g., bottles of wine, are traded for FlexCoins on the blockchain (Step 1). As illustrated in the figure above, this internally issues a mint request, along with an appropriately formatted claim, to the FlexCoin FT smart contract. A representative of the EC operator refreshes the respective management tab and is presented with Carol's pending trade request. Following the inspection of the asset, the trade is approved via the appropriate button, which triggers Steps 8-11 in the blockchain. Lastly, the EC operator executes the trade, which results in Carol's FlexCoin account to be credited with the appropriate amount of FlexCoins. This concludes a simplified description that addresses requirements SF 1.2.1, 1.5 and 1.6. However, we stress that this

serves solely as a toy example for illustration purposes and that a proper solution requires the design and implementation of a separate smart contract that lies outside of the FlexCoin smart contract ecosystem and deals with real-world assets, which are typically modelled as non-fungible tokens (NFTs).

6.3.1.4 Decentralized FT Governance

In previous sections, we provided an overview of the coin minting and burning processes assuming a centralized financial principal, a role which, in practice, is undertaken by the EC operator. In this section, we succinctly discuss an alternative instantiation of the FlexCoin governance smart contract that supports decentralized financial principals, in which the decision-making process is governed by a collection of entities that are part of separate administrative domains. Clearly, decentralized (financial) ecosystems and the processes under which they, ultimately, reach consensus may vary drastically, i.e., there is no "one-size fits all". In the interest of simplicity and brevity, the discussion here will revolve around a simple, voting-based decision-making model, where each entity is allocated a single vote and all votes carry the same weight, i.e., they equally influence the outcome of the consensus process.

On a high-level, one of the main properties of the notary interaction pattern is the expressiveness of the abstractions that it provides between the service and the notary contract. Put simply, it clearly separates the "how", i.e., the code that executes the business logic of an operation, from matters related to the "if" and "when", i.e., the decision-making policies. For instance, in the examples related to the coin minting and burning processes, the FlexCoin FT smart contract is unaware of even the existence of an EC operator. Indeed, it is solely provided with a function call that determines whether it should proceed with the execution of the business logic. For all intents and purposes, there could be an extremely complicated and elaborate process hidden behind that, seemingly, simple function call. This is the main notion upon which we built the scenario described below.

In our toy example, we assume a decentralized financial principal that is comprised by two entities, the EC operator and an EC peer. We assume that in order to approve, or decline a (mint, or burn) request, a majority vote is required which, in our case, requires both parties to submit matching votes. We discuss only the case for minting coins, since that of burning coins follows directly.

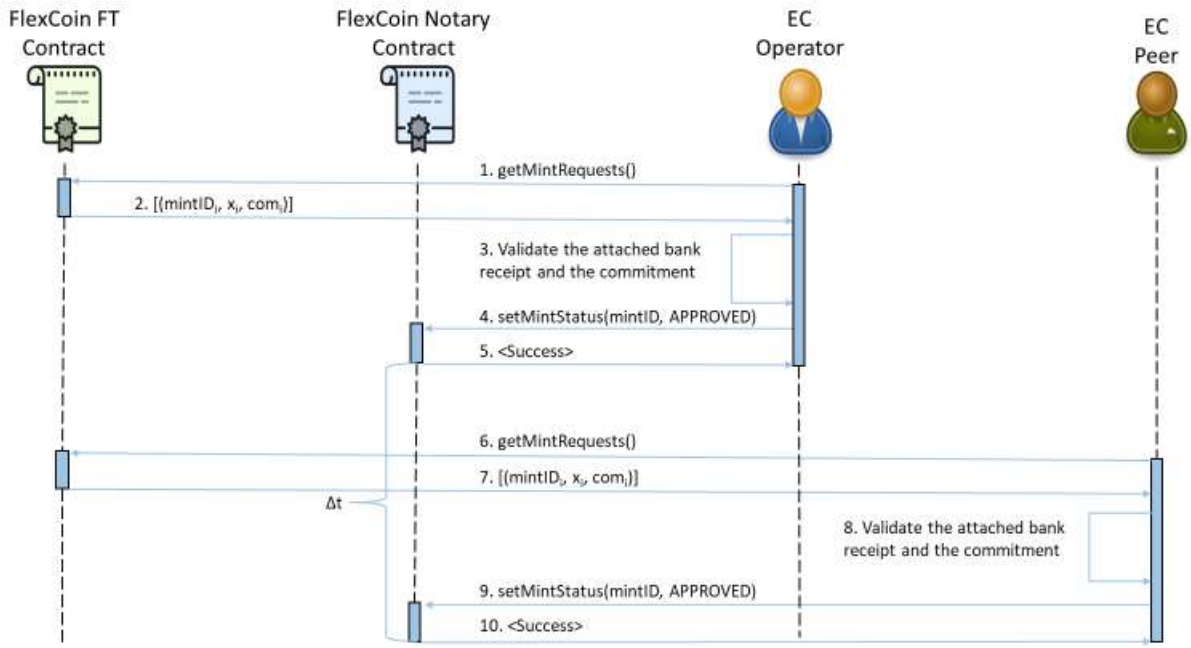

Figure 16: Sequence diagram depicting a simplified instantiation of decentralized financial principle and the involved interactions for approving a mint request.

In Figure 16, we provide a sequence diagram that illustrates the interactions among the entities that

comprise the decentralized financial principal of our toy example and the FlexCoin smart contract ecosystem to approve a, previously issued, mint request. As illustrated, both the EC operator and the EC peer perform exactly the same steps. However, what is interesting, even in this toy example, is the state of affairs in the time window between steps 5 and 10, which for convenience is denoted in the figure as Δt. Any entity, be it the one that issued the mint request, or even one that is part of the decentralized financial principal (assuming a more scaled-up example), attempting to (maliciously) execute the mint request before the successful completion of step 10, will be unable to do so. Indeed, since there is no conclusive majority vote available, the FlexCoin notary contract will output a pending status code for this mint request. Hence, the FlexCoin FT smart contract will not proceed in the execution of the business logic.

Apart from the secure enforcement of the decision-making process, this design also provides for consistent attribution and, as a result, acts as a counterincentive for parties to behave maliciously. Indeed, in an alternative setting where the principal would be comprised by, e.g., three entities (or more), if one of them declined the request, or if it attempted to execute the mint request before a quorum is established, its ledger transaction(s) would serve as evidence to prove its malicious behavior and, thus, would allow for accountability of the malignant entity and, subsequent, penalty enforcement.

6.3.1.5 Holding Coins

The main design goal of the overwhelming majority of FT smart contract proposals in the literature is to expose a set of functionalities that provide for a simplified bank-like, account-based model. Users, or clients, of such smart contracts can create an arbitrary number of accounts, each with its own separate coin balance. An account owner can transfer coins to other destination accounts, which may be owned by other users, i.e., a process similar to that of real-world wire transfers. An additional functionality that is typically supported by FT smart contracts is that of allowances, or approvals, which was introduced by the ERC-20 token standard. Account owners can allow other users to transfer bounded sums of coins from their accounts to other destination accounts that are chosen by the approved users. For instance, Alice, an account owner, can allow Bob, which may not even have an account in the FT smart contract, to transfer, e.g., up to 20 coins, from her account to an account of Bob's choosing. The following clarifications are warranted at this point. First, Bob is not required to transfer the coins from Alice's account in one go, e.g., he can transfer one coin, then transfer another two coins and so on. Second, and most importantly, Bob can only transfer coins from Alice's account if, at that particular point in time, the account's balance is at least equal to the requested amount to be transferred. Put simply, if Alice's account has 3 coins, Bob can only transfer 3 coins at this particular instant. Indeed, the approved coins are not held in a separate pool that locks them away from Alice's main account balance.

At the time of this writing, the functionality provided by FT smart contract proposals is not adequate to serve as a basis for the development of digital marketplaces that are resilient to defrauds. To convey this point more clearly to the interested reader, we provide an intuitive description of a simplified energy trading scenario in which we assume that an ERC-20-compliant smart contract is employed as the underlying common value system. Assume Alice wants to sell 4 kWh of surplus energy from the battery storage system at her residence. To do so, she will post an appropriately formed transaction on the ledger specifying, e.g., a price per kWh sold, among other data points. This will, in turn, act as a signal to interested buyers. In the interest of simplicity, we assume a simple model in which the first entity that successfully posts a purchase transaction for Alice's energy on the ledger will be the declared the winner, to which we will refer as Bob. Even in this very simplified model, there are a number of severe security issues, which we discuss in the following. First, Alice has no guarantee whatsoever that Bob has enough coins to pay for the purchased energy. Second, even if Bob has enough coins, there is no mechanism in place that will force him to pay. Indeed, Bob is by no means forced to transfer any coins from his account to that of Alice. Moreover, even if we assume that we employ the allowance interface of the ERC-20 token standard, again Alice has no guarantee that she will receive the designated number of coins, as Bob can transfer to another account all of his coins prior to approving Alice. Third, Bob has no guarantee that Alice is capable of even delivering the self-advertised amount of energy. For instance, Alice could post multiple transactions for selling the same underlying, physically stored amount of energy, i.e., launch an energy "double-spending" attack (inspired by the formulation of the double-spending problem that is common across all cryptocurrencies). Arguably, Bob should pay based on the

amount of energy that was actually delivered by Alice, which could end up being lower than the amount that was advertised initially for a myriad of reasons. In such cases, depending on the rules that govern the marketplace, it could be the case that Alice would be susceptible to some form of monetary penalty (in units of the underlying FT). However, as was previously highlighted, there is no mechanism in place in the ERC-20 token standard to support such use cases.

Clearly, the situation is far more dire for marketplaces whose financial settlement procedures are more involved and complex. The main underlying issue is that support for monetary commitments was not part of the design process of the ERC-20 token standard. We stress that this line of criticism is not intended to diminish, by any means, the value of this standard, nor the effort invested by the community. Instead, our sole goal is to point out a limitation that, if addressed in a secure and generic fashion, would lead to an even larger adoption of FTs which, as a result, will allow this ingenious technological construct to act as a key-enabler for the development of future digital marketplace ecosystems.

6.3.1.5.1 EIP-1996: Overview

At the time of this writing, the issue of supporting monetary commitments in FT smart contracts has received minimal attention in the literature and the most notable proposal is documented in EIP-1996 [12]. On a high-level, this is a draft proposal for an extension to the ERC-20 token standard that allows coins to be put on hold, i.e., “locking” coin balances to provide guarantees for future transfers. This mechanism bears a great deal of similarity with that of escrows. A meticulous analysis and description of this proposal is out of this document’s scope. However, an overview of the involved actors, as well as, the underlying motivation based on which it was designed is warranted. This will allow us to, subsequently, highlight its limitations and introduce an alternative proposal that provides for increased security and flexibility.

On a high-level, a coin hold specifies the following: 1) a payer, i.e., the entity whose coins will be locked, 2) a payee, i.e., the entity that will ultimately receive the payer’s locked coins (assuming a happy-path scenario), 3) a notary, i.e., an intermediary that oversees the entire financial settlement by deciding the ultimate fate of the held coins and, 4) an (optional) expiration time which, if it expires, allows the coins to be unlocked and, as a result, be returned to the payer’s active account balance. If (4) is not specified, then the coin hold is considered to be perpetual, i.e., there are no temporal restrictions regarding its validity.

The proposal specifies the following operations in regard to coin holds. Execution, which triggers the transfer of tokens and can only be invoked by the designated notary. Coin holds can be partially executed, i.e., it is possible to transfer an amount that is less than what was originally held. In this case, the remaining coins are returned to the payer’s active balance. Release, which returns the held coins to the payer’s active balance. This operation is subject to the aforementioned temporal conditions, assuming an expiration time is specified, otherwise, it can only be invoked by the designated notary.

To facilitate reader understanding, in the following we provide an overview of practical, real-world scenarios that served as the main motivation underpinning this proposal’s design. In the context of regulated tokens, e.g., digital equivalents of real-world fiat currencies, token transfers can proceed gracefully only in cases where all regulatory conditions are satisfied, i.e., a clearing mechanism is in order. Part of the clearing process involves the creation of a coin hold, which will only be executed if all checks pass successfully. Alternatively, in a variety of business applications, payments need to be guaranteed before goods and services can be consumed. In other occasions, apart from the aforementioned guarantees, it might be the case that even the payment’s exact amount is not known beforehand. In the context of the simplified energy trading example that was discussed previously, we demonstrated the relevance of all these aforementioned guarantees in FEVER’s application domain.

6.3.1.5.2 EIP-1996: Critique & Limitations

To the author’s knowledge, EIP-1996 constitutes the most elaborate proposal for guaranteeing financial payments in the context of FT-backed marketplaces. However, it introduces, on the one hand, a series of privacy-related implications and, on the other hand, several limitations, some of which are subtle, but important, strictly from a technical point of view. We dedicate the remainder of this section in discussing

the aforementioned issues.

We first discuss the issue of privacy which, in our view, is the most important one. Recall that the ERC-20 FT standard was designed in the context of public blockchains, or smart contract platforms to be more precise. Since EIP-1996 is a proposed extension to this standard, it follows that it intrinsically inherits the same set of privacy violations regarding its emitted events. More specifically, EIP-1996 introduces a set of events that are related with the lifecycle of coin holds, e.g., HoldCreated, HoldExecuted, among others. The payloads associated with these emitted events provide elaborate information. For instance, the HoldCreated event is accompanied with data that specify the identity of the payer, the payee, the designated notary, the value that is held and the hold's (optional) expiration date. Clearly, this provides significant information to third parties that are not involved in the lifecycle of the coins that are held in regards to the transactions that take place. As a side note, recall our comments regarding the anti-pattern of employing the event mechanism as an analogue of push notifications and its incurred performance penalties. Moreover, the proposal includes various querying APIs (smart contract functions) that allow third parties to dynamically query information related to the lifecycle of a coin hold, such as the notaries associated with it. However, the most critical one, from a privacy perspective, is the "netBalanceOf()" function which, in short, allows third parties to infer, apart from the number of coins held, the account balance of the payer. Arguably, even if it were the case that this function was made available only to the designated notaries, we stress that it would still constitute an important privacy violation as there is no concrete reason for exposing this kind of information to them in the first place. Indeed, recall that the "contract" between the payer, payee and the notary is strictly limited, by definition, in ensuring the payment.

From an architectural and, more generally, technical perspective, EIP-1996 has several drawbacks that, in order for us to highlight them, require some background knowledge regarding Ethereum. In this public smart contract platform, entities, whether they are smart contracts or some external (human) actor, are identified via an Ethereum address. For the purpose of this description, although not technically accurate, one can think of these addresses as a representation of a public key (e.g., as the output of a hash function). Hence, this identifier can be employed by smart contracts to perform, e.g., access control. For instance, in the context of coin holds, only a designated notary is allowed to, e.g., execute a coin hold, since it is the only entity that controls the private key associated with this Ethereum address.

That being said, and as was hinted previously, the proposal allows multiple notaries via the "authorizeHoldOperator()" and the "revokeHoldOperator()" functions, which assign and revoke, respectively, a notary for **all** coin holds related with the instantiation of the FT smart contract. The first issue with this design choice is that it directly couples the implementation of the notary's functionality, i.e., the code for evaluating policies, with that of the FT smart contract. We have already discussed the implications of this point in previous sections and the limitations it imposes. Quite interestingly, we observe a direct manifestation of this limitation, not only in the proposal's design, but also in its reference implementation [13]. Indeed, even if there are multiple notaries specified, a single one can decide whether a coin hold should, e.g., be executed or released. We speculate that the authors of the proposal accounted for multiple notaries to provide for increased liveness and responsiveness. Put simply, to balance the load of requests across multiple notaries. However, this design choice does not allow modelling, e.g., decentralized financial principles.

In our view, the more alarming implication is that a single set of notaries controls **all** coin holds of the underlying FT smart contract. This essentially entails that the notaries are able to implement all policies of all potential marketplaces that are built on top of the FT which is, at minimum, technically intractable. Moreover, even from a business point of view, it is unrealistic to assume that the real-world parties that are behind the technical implementations of the involved notaries have real-world, business relationships.

EIP-1996 introduces a set of status codes in an attempt to capture the individual stages of a coin hold's lifecycle. Examples include individual status codes regarding which party, for instance, released a coin hold, e.g., whether it was the notary, the payee, or even if a hold was released due to its expiry. The approach of providing an exhaustive list of status code regarding the lifecycle of an object (e.g., coin hold) as a basis for reasoning constitutes, from a software development perspective, a well-known bad design choice. More specifically, in the context of smart contracts where the execution of code is expensive, it is preferable to offload code that provides for more elaborate and contextual information

about an object to off-chain software components by leveraging the total ordering and verifiability properties of the ledger.

We conclude with one final, subtle issue of the proposal, which is related with the fact that only the notary can execute, or release, a hold. Notice that the proposal introduces an additional coupling between policy evaluation, or the decision-making process, and access control. Recall that the notary's role is to guarantee that they payment will occur, which does not imply that the notary has to be the only party allowed to, e.g., execute a hold. Apart from the fact that this allows the notary to launch a denial-of-service attack, it hinders interesting future use cases. For instance, in the interest of privacy, the payee might wish to delay, from a temporal perspective, the execution of a hold for a future time where she will have access to, e.g., a virtual private network (VPN), which will provide her with network anonymity. Another interesting future avenue is bundling multiple coin hold related operations in a single transaction, which will effectively provide for what is commonly referred to as "herd anonymity". In the following subsection, we provide an intuitive overview of FlexCoin's coin hold mechanism and its ability to completely resolve all of these challenges by leveraging the notary interaction pattern as its building block.

6.3.1.5.3 FlexCoin Holds Overview

To facilitate reader understanding in regard to FlexCoin's coin hold mechanism, we provide an intuitive description that is based on a simplified, auction-based energy trading scenario. Our toy example accounts for cases where entities can declare their interest in selling excess energy, as well as, buying energy from, potentially competing, sellers. As it will become evident later on in this section, we explore both of these cases, which are modelled as separate auction objects, as it will allow us to demonstrate different aspects, or use cases, of FlexCoin's coin hold mechanism.

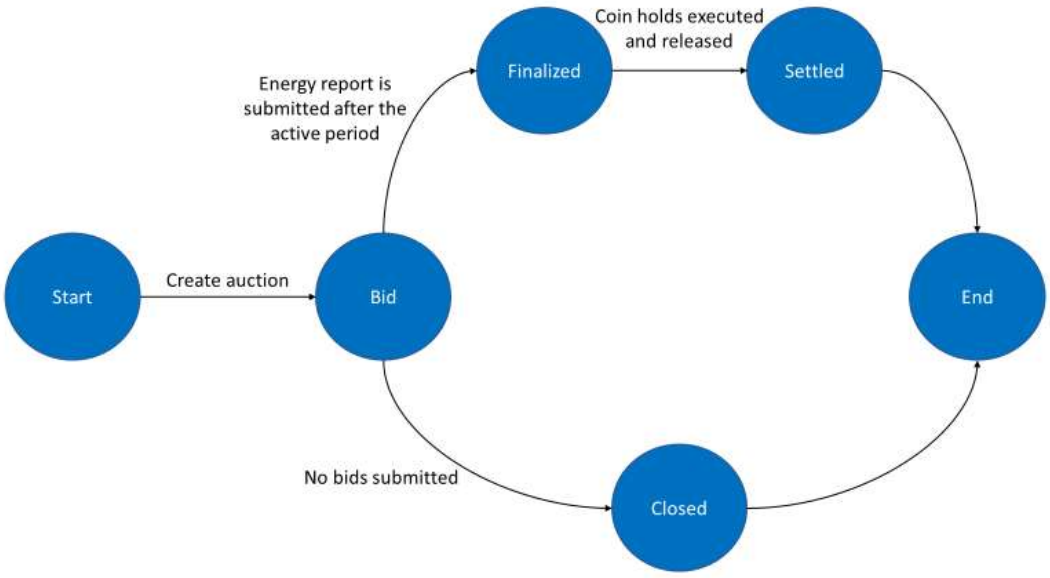

Figure 17: State machine diagram of a simple, auction-based energy trading scenario.

Figure 17 illustrates the lifecycle of an auction object in our example energy trading scenario by means of a state machine diagram, which we describe in the following. The process is initiated by an entity issuing an appropriately formed transaction that expresses its interest in buying, or selling, energy. Relevant data points specified in this transaction could include, among others, an amount of energy, pricing-related information (e.g., units of FlexCoin per kWh) and the active period, i.e., a temporal window during which energy will be physically injected to the underlying grid. Once an auction is created, it immediately transitions to the bidding phase, which allows interested buyers/sellers to declare their participation. Details regarding, e.g., the length of the bidding phase, as well as, rules regarding bidder participation, are implementation-specific and are, thus, out of the context of this discussion. To account

for the case where no bids are submitted, an auction can be closed, which facilitates clean-up of the state of the marketplace smart contract from stale auction objects.

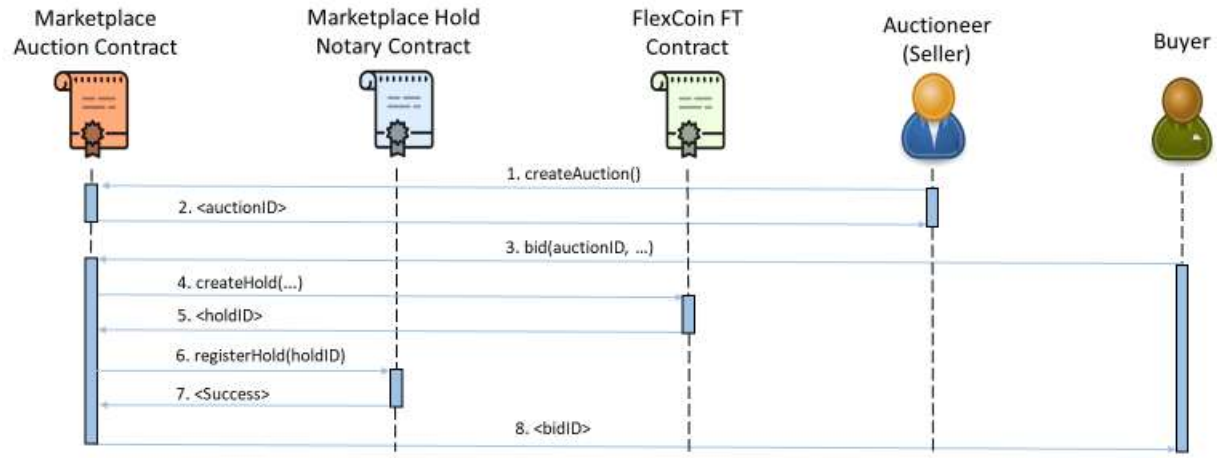

Figure 18: Sequence diagram depicting the creation and subsequent bidding on an auction for selling energy.

In the scenario depicted in Figure 18, an auctioneer, who in this case wishes to sell excess energy, creates an auction object by invoking the marketplace smart contract which, in turn, returns a unique identifier that is associated with this object. We implicitly assume that other actors participating in this trading ecosystem can query the marketplace contract for all auctions that are open for bidding. Based on this, as illustrated in the figure, an interested buyer issues a bid, which internally involves the creation of a hold on her account (Step 4). In our toy example, we assume the existence of a logical entity that acts as the notary in regard to the holds that are created in this particular marketplace. Following the creation of the bidder's hold, the marketplace smart contract registers it to its notary and, subsequently, outputs the identifier of the bid to the buyer.

Following the end of the active period of the auction, we assume that it is finalized by some mechanism that is specific to the marketplace, e.g., by having an oracle submit a report that conveys the amount of energy that was injected in the grid. The marketplace smart contract can employ this report to compute the number of coins that must be paid to the seller, based on the amount that was held from the buyer during the bidding phase, and approve the execution of the hold by appropriately invoking the API of its hold's notary which, once more, is implementation specific.

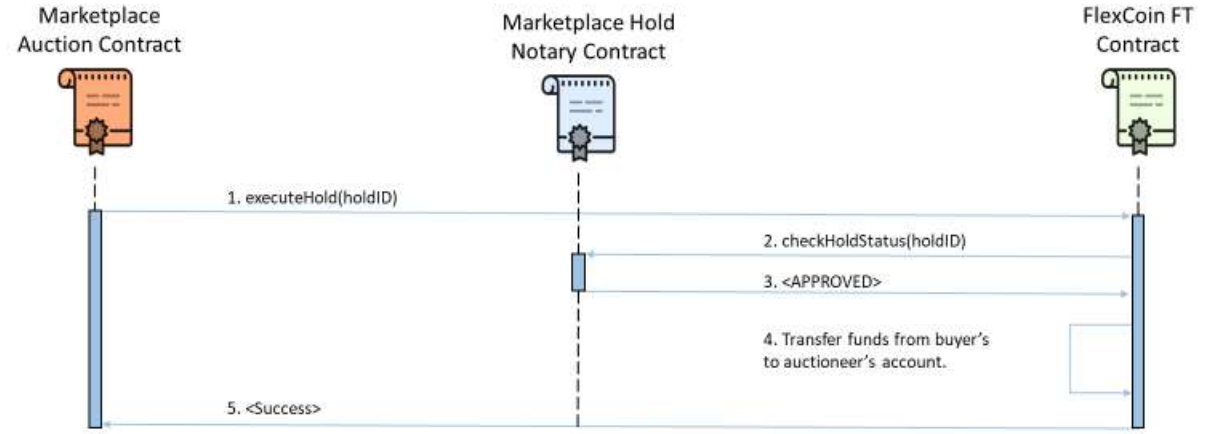

Figure 19: Sequence diagram depicting the execution of the buyer's hold during the auction's settlement phase.

In Figure 19, we assume that the auction's settlement phase is triggered and showcase the interactions involved in executing the buyer's hold. The FlexCoin FT smart contract, as illustrated, checks the status of the hold by querying the marketplace's hold notary contract and, following its approval, transfers the

designated number of coins from the buyer's account to that of the auctioneer. Notice that the depicted flow follows directly from that of the notary interaction pattern, albeit in the context of coin hold operations supported by the FlexCoin FT smart contract.

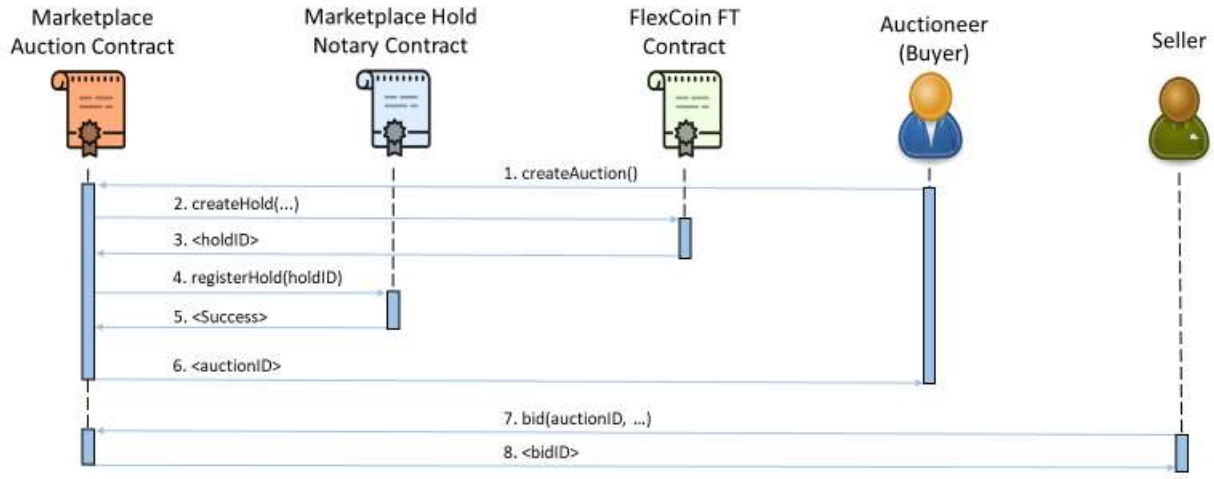

Figure 20: Sequence diagram depicting the creation and subsequent bidding on an auction for buying energy.

In Figure 20, we showcase an alternative scenario in which the auctioneer wishes to buy energy. As illustrated, in this case a hold is created on the auctioneer's account, since it is this actor that acts as the payer. The settlement phase of auctions for buying energy is modified accordingly and, in short, involves the execution of the auctioneer's hold, following the example that was provided previously. In summary, the process described here showcases FlexCoin's ability to address the requirement SF 1.4.

6.4 FlexCoin Governance Smart Contract

In this section, we present the technical specification of FlexCoin's governance smart contract, which, on the one hand, abstracts the intricacies that govern the creation and conversion of value between the blockchain and the real-world and, on the other hand, provides a generic interface that can support both centralized and decentralized financial principals, both of which are addressed by building on top of the notary interaction pattern.

The technical specification of this smart contract is organized as follows. In Section 6.4.1, we present the interface, i.e., the functions that the FlexCoin governance smart contract exposes to the outside world, be it other contracts or external actors (e.g., clients of the HLF network). In Section 6.4.2, we specify, via UML class diagrams, the data model of this smart contract's interface. Lastly, in Section 6.4.3, we present a proposal for an HTTP abstraction of this smart contract's functionality that can be leveraged for, e.g., smooth integration with web technologies.

6.4.1 Interface

In Table 5, we provide a list of the functions that are exposed by the FlexCoin governance smart contract. For each of its exposed functions, we specify its input and output objects (if any) along with the enforced access control policies. Details regarding the input and output objects are provided in the next section. The "Only Configured Principal" policy entails that this function can only be invoked by one of the financial principals with which the smart contract has been initialized as it is instantiated on the ledger, based on the configuration object that is input to the smart contract's Init() function.

The term "n/a", when used in the context of function inputs, entails that it does not accept any arguments. Correspondingly, in the context of function outputs, it entails that the smart contract does not return a particular result, apart from the event of its (un)successful invocation, which in HLF is part of the peer's response to endorsement requests from clients.

We employ the notation *Object.Attribute* to denote the context of a function's input by linking it to the

data model specification of the following section. For instance, the CheckMintStatus() receives as input the identifier attribute of a MintBurn object. Objects that are input to functions are assumed to be JSON encoded (same holds for returned objects).

Table 5: Specification of the interface of the FlexCoin Governance smart contract where for each exposed function its inputs, outputs and access control policies (if any) are provided.

Function	Input	Output	Access Control Policy
Init()	FlexCoinGov Config	n/a	n/a
SetMintStatus()	MintBurnStateMod	n/a	Only Configured Principal
CheckMintStatus()	MintBurn.ID	NotaryStatusCode	n/a
SetBurnStatus()	MintBurnStateMod	n/a	Only Configured Principal
CheckBurnStatus()	MintBurn.ID	NotaryStatusCode	n/a
CheckPrincipalStatus()	n/a	NotaryStatusCode	n/a
GetFlexCoinPrincipals()	n/a	Principal[]	n/a

A concise description of the functionality of each of the functions that this smart contract exposes is as follows:

- **Init():** Initialization function that is invoked only once, i.e., during the instantiation of the smart contract on the ledger, which receives and, subsequently, stores in its internal state its configuration parameters.
- **SetMintStatus():** Attempts to execute the state transition function for mint requests based on the input payload.
- **CheckMintStatus():** Notary function for mint requests.
- **SetBurnStatus():** Attempts to execute the state transition function for burn requests based on the input payload.
- **CheckBurnStatus():** Notary function for burn requests.
- **CheckPrincipalStatus():** Notary function for access control policies regarding privileged functions exposed by the FlexCoin FT smart contract.
- **GetFlexCoinPrincipals():** Accessor function that returns identification information of the financial principals with which this smart contract is configured.

6.4.2 Data Model

In this section, we introduce and discuss the data model of the FlexCoin governance smart contract, which is presented in Figure 21 via UML class diagrams.

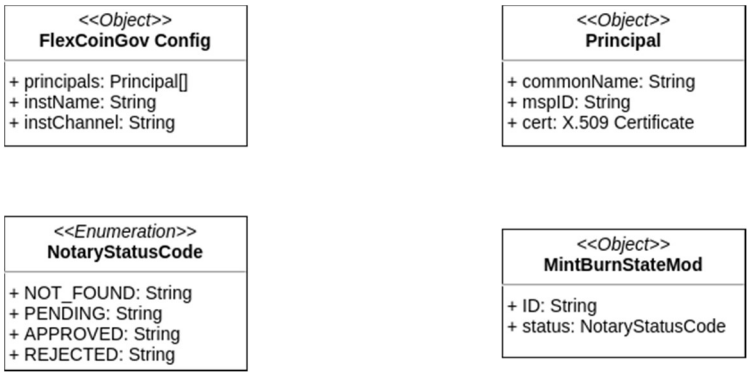

Figure 21: Simplified UML class diagram of the objects that are input/output to/from the functions of the FlexCoin Governance smart contract.

The FlexCoinGov Config object serves as the starting point of our data model discussion, as it is input to the smart contract's initialization function. It encompasses two parameters, instName and instChannel, which convey to the smart contract its instantiation name and channel, respectively. Lastly, it includes a list of objects, which include the necessary identification information for the financial principals with which it is configured.

The status codes involved in the notary interaction pattern have already been introduced in a previous section, however, in the interest of completeness and clarity, are depicted in the figure above as an enumerated string type. Lastly, the MintBurnStateMod object encompasses the identifier of the referenced mint, or burn, request whose status is subject to change.

6.4.3 HTTP API Abstraction

In this section, we introduce a proposal for an HTTP API abstraction of the functions that are exposed by the FlexCoin governance smart contract, which is specified in Table 6. Each line of the table specifies the name of the endpoint, its request type, its parameters, the payload that is returned in the response's body (if any) and, lastly, a short description that specifies the function of the smart contract that is invoked.

Table 6: Specification of HTTP API abstraction of the functions exposed by the FlexCoin Governance smart contract.

Endpoint	Parameters	Return	Description
/burn/status **POST**	• Authentication or ID token (implementation specific) • JSON encoded MintBurnStateMod object in the body of the request	n/a	Invokes the function SetBurnStatus() with the provided inputs.
/burn/status/{burnID} **GET**	• Authentication or ID token (implementation specific) • Identifier of the burn request	NotaryStatusCode as a string in response body	Invokes the function CheckBurnStatus() with the provided inputs.
/mint/status **POST**	• Authentication or ID token (implementation specific) • JSON encoded MintBurnStateMod object in the body of the request	n/a	Invokes the function SetMintStatus() with the provided inputs.
/mint/status/{mintID} **GET**	• Authentication or ID token (implementation specific) • Identifier of the mint request	NotaryStatusCode as a string in response body	Invokes the function CheckMintStatus() with the provided inputs.

/principals **GET**	• Authentication or ID token (implementation specific)	JSON array of Principal objects in response body.	Invokes the function GetFlexCoinPrincipals().

As specified in the parameter column of Table 6, the proposed API needs to be protected from unauthorized use. Depending on the concrete implementation of the API and its complementary security infrastructure (if any), as well as, the physical storage of private keys, among others, this can be achieved by, e.g., simple authentication, or even ID tokens that are based on widely deployed, industry standards, such as JSON Web Tokens (JWTs). We will elaborate more on this matter following the initiation of the implementation phase.

6.5 FlexCoin FT Smart Contract

In this section, we present the technical specification of FlexCoin's FT smart contract, whose main purpose is to provide a common value substrate that can be used as a building block for developing arbitrary marketplaces that involve the digital equivalent of financial commitments of real-world trading scenarios.

The technical specification of this smart contract is organized as follows. In Section 6.5.1, we present the interface, i.e., the functions that the FlexCoin FT smart contract exposes to the outside world, be it other contracts or external actors. In Section 6.5.2, we specify, via UML class diagrams, the data model of this smart contract's interface. Lastly, in Section 6.5.3, we present a proposal for an HTTP abstraction of this smart contract's functionality that can be leveraged for, e.g., smooth integration with web technologies.

6.5.1 Interface

In Table 7, we provide a list of the functions that are exposed by the FlexCoin FT smart contract. For each of its exposed functions, we specify its input and output objects (if any) along with the enforced access control policies. Details regarding the input and output objects are provided in the next section. The "Only FlexCoinGov Principal" policy entails that this function can only be invoked by one of the financial principals with which the FlexCoin governance smart contract has been initialized as it is instantiated on the ledger. The "Only Account Owner" policy entails that this function can only be invoked by the owner of a specific FlexCoin account. In terms of inputs and outputs, we employ the same set of conventions as those of Section 6.4.1.

Table 7: Specification of the interface of the FlexCoin FT smart contract where for each exposed function its inputs, outputs and access control policies (if any) are provided.

Function	Input	Output	Access Control Policy
Init()	FlexCoinFT Config	n/a	n/a
CreateAccount()	n/a	Account	n/a
GetAccount()	Account.ID	Account	Only Account Owner
GetAccounts()	n/a	Account[]	n/a
GetAccountTxHistory()	Account.ID	AccountLogEntry[]	Only Account Owner
Transfer()	Transfer	n/a	Only Account Owner

CreateHold()	Hold	Hold.ID	Only Account Owner
ExecuteHold()	HoldExecution	NotaryStatusCode	n/a
ReleaseHold()	Hold.ID	NotaryStatusCode	n/a
RequestMint()	MintBurn	MintBurn.ID	Only Account Owner
ExecuteMint()	MintBurn.ID	NotaryStatusCode	n/a
ReleaseMint()	MintBurn.ID	NotaryStatusCode	n/a
GetMintRequests()	n/a	MintBurn[]	Only FlexCoinGov Principal
GetMintHistory()	n/a	MintBurnLogEntry[]	Only FlexCoinGov Principal
RequestBurn()	MintBurn	MintBurn.ID	Only Account Owner
ExecuteBurn()	MintBurn.ID	NotaryStatusCode	n/a
ReleaseBurn()	MintBurn.ID	NotaryStatusCode	n/a
GetBurnRequests()	n/a	MintBurn[]	Only FlexCoinGov Principal
GetBurnHistory()	n/a	MintBurnLogEntry[]	Only FlexCoinGov Principal

A concise description of the functionality of each of the functions that this smart contract exposes is as follows:

- **Init():** Initialization function that is invoked only once, i.e., during the instantiation of the smart contract on the ledger, which receives and, subsequently, stores in its internal state its configuration parameters.
- **CreateAccount():** Creates a new FlexCoin FT account which is owned by the invoking user.
- **GetAccount():** Returns a description of the status of a FlexCoin FT account.
- **GetAccounts():** Returns a description of all the FlexCoin FT accounts (if any) that are owned by the invoking user.
- **GetAccountTxHistory():** Returns an ordered event log of all the actions related to a particular FlexCoin FT account.
- **Transfer():** Wires tokens between FlexCoin FT accounts.
- **CreateHold():** Creates a new hold on the invoking user's account, i.e., a number of coins are reserved on the side.
- **ExecuteHold():** Executes the hold specified by the input payload, assuming the designated notary approves it.
- **ReleaseHold():** Release the hold specified by the input payload, assuming the designated notary approves it.
- **RequestMint():** Entry point for account owners to request the minting of coins on one of their accounts.
- **ExecuteMint():** Coins are minted to a FlexCoin FT account, assuming the approval of the FlexCoin governance smart contract.
- **ReleaseMint():** A rejected mint request is moved to the respective historical log of requests.
- **GetMintRequests():** Accessor for pending mint requests issued by FlexCoin account owners that can be examined by the financial principals of the FlexCoin FT governance smart contract

to decide upon.

- **GetMintHistory():** Outputs the history of all mint requests that have reached a terminal state of their state transition function, i.e., approved or rejected.
- **RequestBurn():** Entry point for account owners to request the burning of coins from one of their accounts.
- **ExecuteBurn():** Coins are burned from a FlexCoin FT account, assuming the approval of the FlexCoin governance smart contract.
- **ReleaseBurn():** A rejected burn request is moved to the respective historical log of requests.
- **GetBurnRequests():** Accessor for pending burn requests issued by FlexCoin account owners that can be examined by the financial principals of the FlexCoin FT governance smart contract to decide upon.
- **GetBurnHistory():** Outputs the history of all burn requests that have reached a terminal state of their state transition function, i.e., approved or rejected.

6.5.2 Data Model

In this section, we introduce and discuss the data model of the FlexCoin FT smart contract, which is presented in Figure 22 via UML class diagrams.

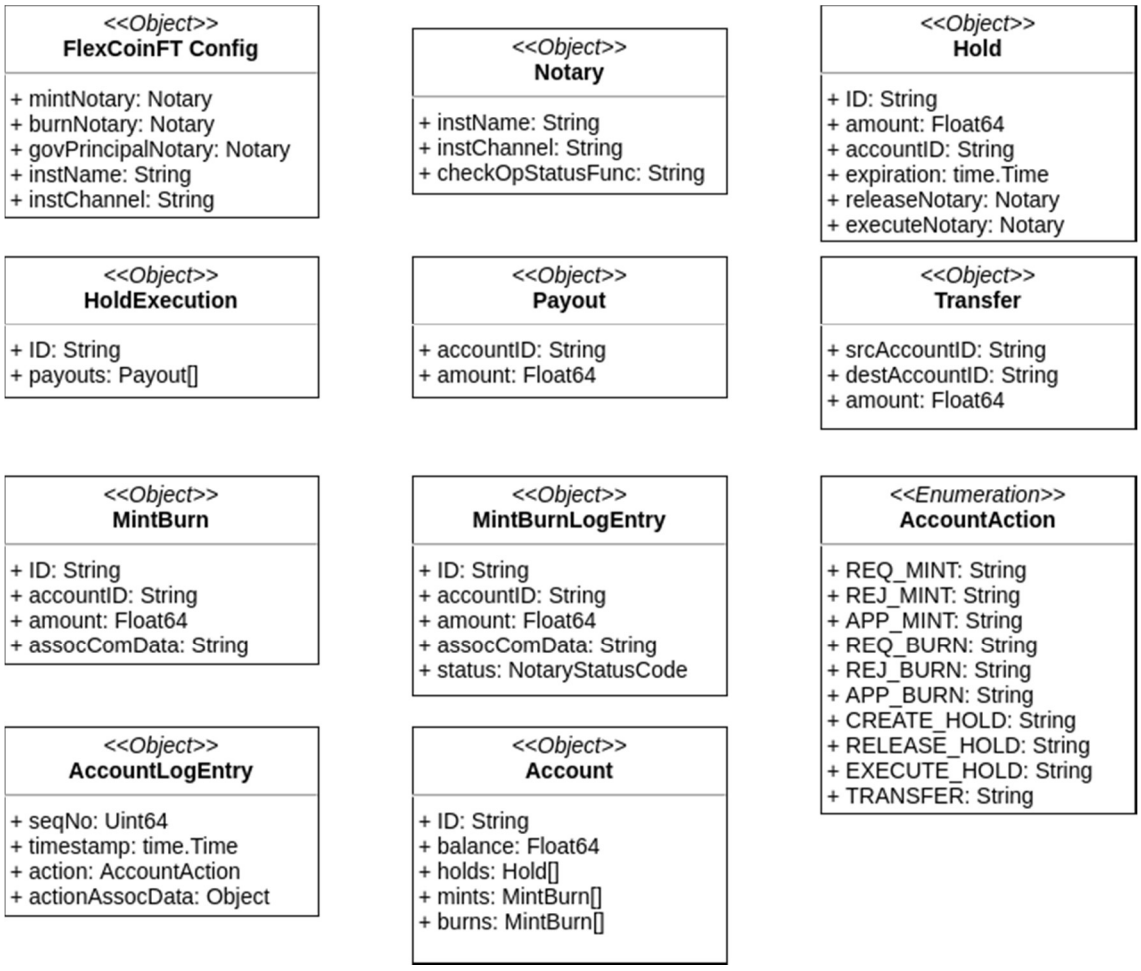

Figure 22: Simplified UML class diagram of the objects that are input/output to/from the functions of the FlexCoin FT smart contract.

We begin our description from the FlexCoinFT Config object, which is input to this smart contract's initialization function during its instantiation on the ledger. As illustrated in the figure above, this configuration object encompasses the necessary information in regards to the notaries of mint and burn requests, as well as, the notary that implements the access control policies for this contract's privileged operations. Lastly, as in the case of the FlexCoin governance contract, the instantiation name and channel are also part of this contract's configuration object.

The Hold object encodes all the necessary information related to a coin hold, i.e., its unique identifier, the number of reserved coins, the identifier of the FlexCoin account with which it is associated, an optional expiration time and the notaries that govern its potential release or execution. Note that a hold's identifier is assigned and returned by the FlexCoin FT smart contract during its creation, i.e., following the successful invocation of the CreateHold() function. The HoldExecution payload encompasses the identifier of the coin hold with which it is associated and an array of pay-outs, which specify the distribution of the held coins across FlexCoin accounts.

The Transfer object encodes the number of coins (amount) to be transferred from the source to the destination account, which are denoted by their identifiers, i.e., srcAccountID and destAccountID, respectively.

The MintBurn object, as is hinted by its name, encodes the information that are required to submit a mint or burn request, which are effectively the same. As in the case of coin hold identifiers, note that identifiers related to mint or burn requests are returned by the FlexCoin FT smart contract following their successful submission on the ledger. A MintBurnLogEntry is a simple extension of a MintBurn object, which encompasses the terminal status of the mint or burn request.

The Account object provides a description of the current status of a FlexCoin account. It includes the unique identifier of the account, its active balance, as well as, arrays denoting the coin holds, mint and burn requests that are in pending status.

Lastly, an AccountLogEntry object encompasses information related to a particular action that has taken place in the context of a specific account. These are totally ordered based on incrementing sequence numbers (seqNo), timestamped based on the value included in the transaction's proposal, classified according to the AccountAction enumerated type and, finally, accompanied with an abstract data type (actionAssocData) that can be used to encode additional contextual information related with the undertaken action.

6.5.3 HTTP API Abstraction

Similar to the corresponding section of the FlexCoin governance smart contract, in this section, we introduce a proposal for an HTTP API abstraction of the functions that are exposed by the FlexCoin FT smart contract, which is specified in Table 8.

Table 8: Specification of HTTP API abstraction of the functions exposed by the FlexCoin FT smart contract.

Endpoint	Parameters	Return	Description
/account **POST**	• Authentication or ID token (implementation specific).	JSON representation of Account object in response body	Invokes the function CreateAccount().
/account/{accID} **GET**	• Authentication or ID token (implementation specific). • Identifier of user's FlexCoin account.	JSON representation of Account object in response body	Invokes the function GetAccount() with the provided input.
/account/all **GET**	• Authentication or ID token (implementation specific).	JSON array of Account objects that are owned by the invoking user	Invokes the function GetAccounts().

/account/log/{accID} **GET**	• Authentication or ID token (implementation specific). • Identifier of user's FlexCoin account.	JSON array of AccountLogEntry objects in response body.	Invokes the function GetAccountTxHistory() with the provided input.
/account/transfer **POST**	• Authentication or ID token (implementation specific). • JSON encoded Transfer object in the body of the request.	n/a	Invokes the function Transfer() with the provided input.
/hold **POST**	• Authentication or ID token (implementation specific). • JSON encoded Hold object in the body of the request.	n/a	Invokes the function CreateHold() with the provided input.
/hold/execute **POST**	• Authentication or ID token (implementation specific). • JSON encoded HoldExecution object in the body of the request.	String representation of NotaryStatusCode	Invokes the function ExecuteHold() with the provided input.
/hold/release/{holdID} **POST**	• Authentication or ID token (implementation specific). • Identifier of the hold whose release will be attempted.	String encoded NotaryStatusCode.	Invokes the function ReleaseHold() with the provided input.
/mint **POST**	• Authentication or ID token (implementation specific). • JSON encoded MintBurn object in the body of the request.	n/a	Invokes the function RequestMint() with the provided input.
/mint/execute/{mintID}	• Authentication or ID token	String encoded	Invokes the function ExecuteMint() with the

POST	(implementation specific). • Identifier of the mint request.	NotaryStatusCode.	provided input.
/mint/release/{mintID} **POST**	• Authentication or ID token (implementation specific). • Identifier of the mint request.	String encoded NotaryStatusCode.	Invokes the function ReleaseMint() with the provided input.
/mint/requests **GET**	• Authentication or ID token (implementation specific).	JSON array of MintBurn objects in response body.	Invokes the function GetMintRequests().
/mint/history	• Authentication or ID token (implementation specific).	JSON array of MintBurnLogEntry objects in response body.	Invokes the function GetMintHistory() with the provided input.
/burn **POST**	• Authentication or ID token (implementation specific). • JSON encoded MintBurn object in the body of the request.	n/a	Invokes the function RequestBurn() with the provided input.
/burn/execute/{burnID} **POST**	• Authentication or ID token (implementation specific). • Identifier of the burn request.	String encoded NotaryStatusCode.	Invokes the function ExecuteBurn() with the provided input.
/burn/release/{burnID} **POST**	• Authentication or ID token (implementation specific). • Identifier of the burn request.	String encoded NotaryStatusCode.	Invokes the function ReleaseBurn() with the provided input.
/burn/requests **GET**	• Authentication or ID token (implementation specific).	JSON array of MintBurn objects in response body.	Invokes the function GetBurnRequests() with the provided input.
/burn/history **GET**	• Authentication or ID token (implementation specific).	JSON array of MintBurnLogEntry objects in response body.	Invokes the function GetBurnHistory() with the provided input.

7 FlexTrading DAPP Specification

This section focuses on the FlexTrading DAPP and provides a detailed specification of functionalities to fulfill the requirements, specified in Sections 3.3.1, 3.3.2, and 3.3.3. The overall objective of the FlexTrading DAPP is to provide a P2P marketplace in scenarios when no trusted market operator can be assumed. The intent is not to provide a fundamentally new concept of the P2P marketplace, but instead adapt and combine existing state-of-the-art (P2P) energy and flexibility marketplace techniques to (1) support the FlexOffer concept, (2) cater P2P trading scenarios, i.e., the user stories US1-2 and US4-5 from Section 0, and (3) fulfill the requirements from Section 3.3.3. This section will start discussing FlexOffers as (potential) products that need to be supported by FlexTrading DAPP.

7.1 FlexOffers as Products

FlexOffers are adopted in FEVER as a common unified representation of energy flexibilities, enabling FEVER system interoperability and the reuse of flexibility management algorithms, libraries, and tools. By following this general adoption of FlexOffers and the requirement ST1.4 (Section 3.3.3), FlexOffers are to be used to define both selling and buying bids of the P2P marketplace (FlexTrading DAPP). We now discuss the two types of products, relevant in FEVER, which can be specified by a FlexOffer (FO):

1. **P_E: Energy Product** - a series of electricity amounts (in kWh) consumed/produced within consecutive time intervals (e.g., of 15 min duration). The full series can, potentially, be shifted in time within the user-given bounds (during matching) – to improve the buyer's or seller's position (utility) – e.g., see US1 in Section 2.1.

2. **P_F: Energy Flexibility Product** - a series of electricity amount deviations (deltas) that represent *increased* or *reduced* production or consumption (of a prosumer, in ∆kWh) within consecutive time intervals (e.g., of 15 min duration). The full series can, potentially, be shifted in time within the user-given bounds (during matching) – to improve the buyer's or seller's position (utility).

For simplicity, these product instances below in the text are called **P_E** and **P_F** products.

A single peer can offer and/or request one of these products by submitting a *bid, i.e., a* FlexOffer. Normally, this should be done in an automatic fashion through an agent system - FSPA or FSCA. In both cases, *FlexCoins* (Section 3.2) are used to define the value of P_E and P_F. The products P_E and P_F are separated and traded in two different market pools, offered by FlexTrading DAPP. This, FlexTrading DAPP acts as a multi-product marketplace. Product P_E-P_F FOs are indivisible (i.e., are "take it or leave it"). This means that energy amounts and prices of **all** FO slices are decided by P2P-FTP (FlexTrading DAPP) if a bid is accepted. This property means that a peer will be able to guarantee/plan power delivery ahead in time for the full duration of a process in question (e.g., EV charging, washing dishes, etc.).

A single FO defines P_E or P_F using *time*, *energy amount*, *location*, and *price* parameters and capture complex inter-dependencies between them. We now discuss how exactly these products are represented by FlexOffers.

7.2 FlexOffers as Energy Products

Figure 23 exemplifies a single FO, representing P_E.

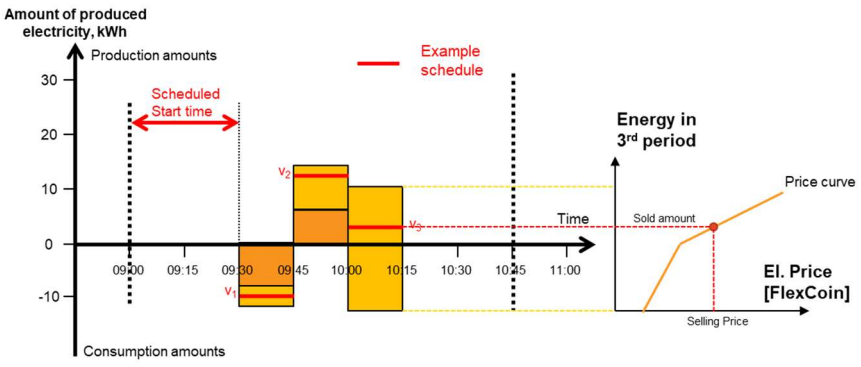

Figure 23. Example FlexOffer with price information (for 3rd slice) for P_E

In this example, a peer expresses electricity demand and supply amounts (in kWh) for 3 consecutive time intervals (of 15mins), denoted as *slices*. The starting time of energy delivery (and consumption) is not fixed and can be varied within the permitted FO *time flexibility* interval, specifically from 09:00 (*earliest start time*) to 10:00 (*latest start time*) with 10:45 as the *latest end time* (see black dashed lines). The offered/requested energy amounts of these 3 slices are also not fixed and can be varied within the respective *energy flexibility* intervals (yellow areas). The first 2 consecutive slices additionally define minimum energy amounts to be consumed (1st) and generated (2nd), respectively (orange areas). Each such interval also defines a *price curve*, which specifies the concrete amount of energy a peer wishes to buy or sell as a function of energy price (in FlexCoins) he/she will be offered for this interval. When submitted, FlexTrading DAPP will match this bid-FO with bid-FOs from other peers, by exploring a variety of allocation (start time) options. Ultimately, P2P-FTP will generate a *schedule*, which specifies concrete demand and supply *start time* (red arrow), *energy amounts* (v_1-v_3), and *selling/buying price* (in FlexCoin) for each time interval.

For P_E, no seller-side and buyer-side bids are distinguished. Based on a bid submitted, FlexTrading DAPP will automatically decide whether a particular peer acts as a buyer or a seller (of electricity) at a particular time interval.

Unlike bids in the traditional auction-based electricity markets, a FO may capture complex inter-dependencies between slices (time units), which makes P_E fundamentally different and incompatible with the majority of solutions discussed in the literature. When bid FOs have only 1 slice each, trading in the P_E marketplace provided by the FlexTrading DAPP is similar to a double auction with sealed bids.

7.2.1 FlexOffers as Energy Flexibility Products

The product P_F is modelled in a similar way, but with a few significant differences and additions in FO parameters. Specifically, FOs of the P_F product define so-called *baseline schedules* and *reward* attributes, shown in Figure 5. The baseline *schedule* (green lines) indicates a socio-economically optimal electricity consumption/generation schedule of a peer, whereas *reward* attributes indicate possible deviations from this baseline depending on a *reward* offered to the peer for such a *flexibility service*. For example, *energy shift reward* for the FO slice 3 in Figure 24 indicates the peer's possibility to further increase or reduce production in the range from 7 to 30kWh depending on the reward offered, with the baseline value being 15kWh. Likewise, a peer may indicate *time shift reward* that enables several time-shifting options within the full FO time shifting period (see yellow lines at the bottom) depending on the *time shifting reward* offered. Other types of rewards are possible, e.g., *commitment reward* which indicates the peer's requested amount of FlexCoins for fulfilling a prescribed schedule (irrespective to whether it represents a baseline or a deviation). Note, without such a commitment reward otherwise, a peer is allowed to consume/produce electricity in any desirable way.

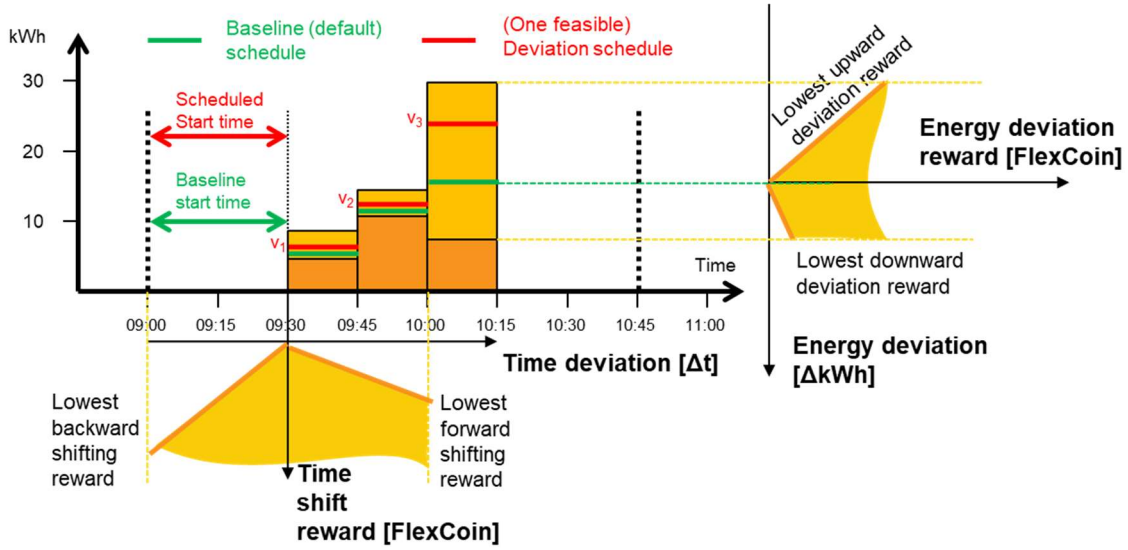

Figure 24. Example FlexOffer with price information (for 10:00-10:15) for P_F

For P_F, buyers and sellers are distinguished. This means that the sender of a FO (bid) needs to indicate whether he/she *requests* or *offers* such a flexibility service.

7.2.2 EC Peer Bidding Strategy

The FO representation of bids give a wide range of options for peers to specify P_E and P_F products. FOs can be generated (selectively) for a single or multiple consecutive time intervals (slices), with or without minimum (/maximum) requested total and/or interval energy, simple or advanced price curves and/or reward parameters. A specific approach of a FO generation is specified manually by the user as a *bidding strategy (settings)*.

7.3 P2P Marketplace Concept

In this section, a conceptual idea of the PE and PF marketplace is described. We will start with the market organization (structure), and continue with market clearing, pricing, and settlement mechanisms.

7.3.1 Market Organization

The market for products PE and PF is organized as a number of distinct independent groups (bidding areas), each representing a physical EC (e.g., one for SWW, one for SWH). The groups are considered flat (i.e., following the “copper-plate” model), which means that neither grid capacity limits nor grid fees and costs of transferring energy between group peers are considered. A peer may trade in one or more such groups. For trading in a specific group, a peer (user) has to hold a digital contract between the peer and an EC (group) stored on the blockchain. Such a contract specifies terms and conditions applicable in this specific group - set by the EC operator (e.g., allowed imbalances and their fees). Each traded PE and PF FO should include a (spatial) parameter indicating the concrete market group (bidding area) the product is traded for. Such groups and contracts are configured using the Community (Identity) Management (see Section 5) and FlexTrading DAPPs, respectively.

7.3.2 Market Clearing Mechanism

FlexTrading DAPP continuously receives P_E-P_F FOs and groups them based on a product type and a targeted EC. Finally, within each group, the system matches them by repeatedly solving a complex optimization problem, which tries to find the best techno-economical solution (see *Pricing* below) within some time horizon of T. When solving this problem, total energy and cost balance constraints, as well as price curves (and reward attributes) and inherent FO constraints of all participating peers are considered. To avoid infeasibilities when solving this optimization problem, the EC Operator includes so-called *default bids* (FOs), which represent amounts of energy imported from the grid (/exported to the grid) to EC (less favorable). Clearing/contracting, however, is performed when no further (re-)scheduling of a FO is allowed – this is explicitly indicated by the FO’s *latest assignment time* parameter. Contracted energy amounts are typically defined for fixed quarter-aligned time intervals (e.g., 1:15 to 1:30) - thus matching FO slice specifications. This overall process is visualized in Figure 25.

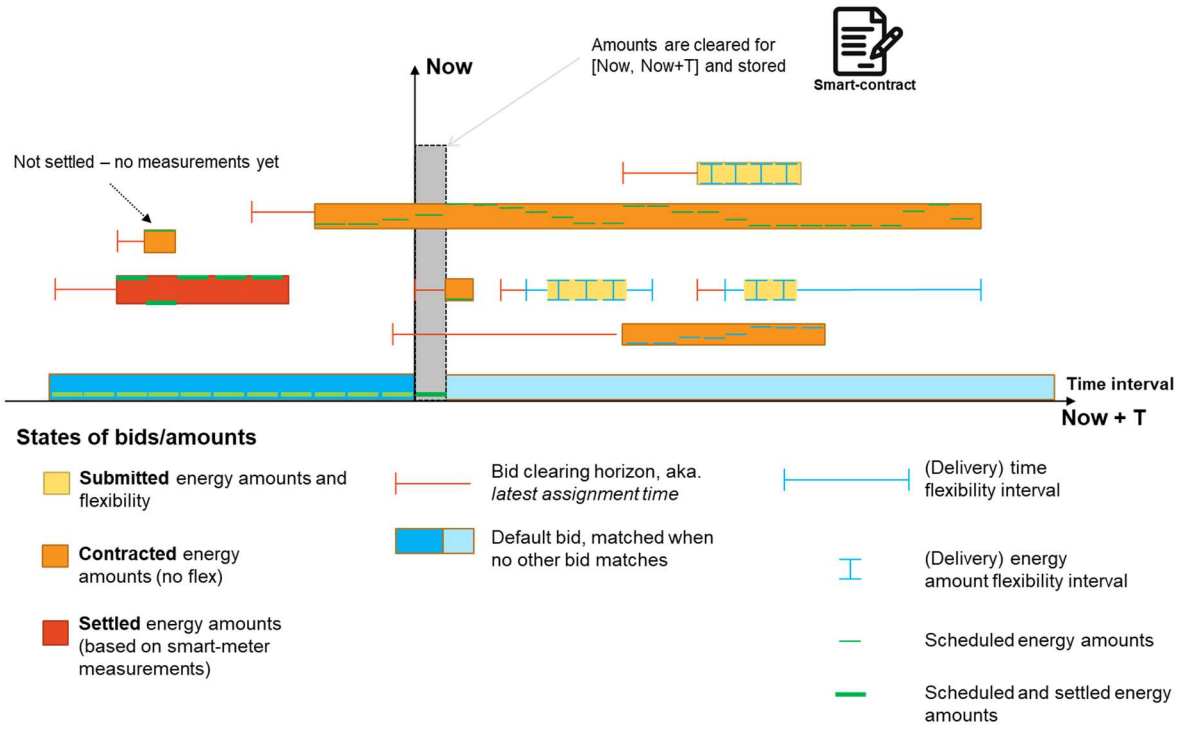

Figure 25 FlexTrading DAPP market clearing mechanism

As seen in Figure 25, a received FO (bid) goes through 3 states:

- **Submitted** bids are those which were checked, accepted, and scheduled (potentially several times). Bids remain in this submitted state until the (inside the FO) indicated *latest assignment time* is due. This FO parameter is peer (user)-specified, and often matches the value of *earliest start time* for quickly dispatchable loads.
- **Contracted** bids can no longer be (re-)scheduled. The last computed FO schedule is considered final and is written into the DLT ledger. Thus, it becomes the subject of the peer's (closed) contract and (the FSPA/xEMS of) the peer is required to deliver and consume energy amounts according to this contracted schedule.
- **Settled** bids are in their final state of the FO lifecycle. Required funds (of FlexCoins) are transferred to/from the peer's *personal account* into a *market pool account* (due to *peer-to-community* contracting) managed by the *EC operator.* These transactions are fully automated (since they are governed by DLT smart-contracts). A bid enters the *settled* state either if measurements from registered smart-meters (main or sub-meters) are received for the contracted FO period, or no measurements from a peer (meter) are available by a predefined deadline (which renders a delivery failure).

In its nature, our proposed market concept is a variant of continuous auction model, where a marketplace tries to clear the market each time a new order is placed, instead of clearing at discrete time instances.

7.3.3 Pricing

For each market group (bidding area), P2P-FTP allows EC operators to configure the following pricing-related parameters:

- **Pricing mechanism** - defines how prices of P_E products are established:
  - *Pay-as-bid* - a peer gets its offered price (for a specific time interval);
  - *Pay-as-cleared* (aka "uniform pricing") - all peers receive the same price (market-clearing price), even if they have offered different prices.
- **Energy source ranking** - defines how available energy sources (and sinks) are ranked during the matching process. By default, *merit order* (sorting by price in FlexCoins) is used. However, sorting based on energy sources are possible, i.e., that some sources of electricity (e.g., from

RES) are prioritised and some are penalized (e.g., fossil-based) inside a EC.

7.3.4 Settlement Process

After a FO is executed and energy is consumed and produced for the complete duration of a FO, measurements from registered and EC-Operator smart-meters (or peers) are collected by P2P-FTP (through FSPAs/FSCAs). Based on these measurements, peer bids are settled and funds (in FlexCoin) are automatically transferred between participants. In the case of imbalances (differences between scheduled and measured amounts), the FlexTrading DAPP will automatically provision surplus or deficit energy from/to a back-up energy source (e.g., the grid, less favourable). Also, pre-defined penalties (in FlexCoin, specified in the contract) are automatically applied to the peer, while also reducing the peer's performance index. This performance index can, potentially, be considered during energy source ranking (see Pricing above) and can increase or reduce the peer's trading position (for getting better prices for PE and PF products). Finally, EC-Operators will be provided a view of measurements from different EC peer meters within trading periods so electricity amounts traded in the P2P marketplace are excluded (not accounted for) in the final electricity supplier's bill (ST1.8, Section 3.3.3).

7.4 P2P Marketplace Specification

In this section, we discuss how the previously presented P2P marketplace concept should be realized on top of the HLF in the FlexTrading DAPP.

7.4.1 Actors

As presented earlier, two essential business actors (roles) are involved in P2P trading *EC Operator* and *EC Peer*. We make the following assumptions about them:

- **EC Operator** has no individual business interests in P_E and P_F P2P trading. Instead, it aims at helping an EC as the whole and aims at facilitating trading between EC peers. It thus manages independent *auctions* for P_E, P_F, or both types of products, and grants or revokes access of various peers to a particular auction (see ST1.1 – ST1.3, Section 3.3.3).
- **EC Peer** is a party connected to the EC grid, consuming and producing electricity. It has a valid FlexCoin account (see Section 0) and seeks both private (e.g., save money) as well as communal objectives (e. g., maximise the EC's self-consumption). EC Peer is equipped with a number of *power meters* (smart or sub-meters) and is willing to share their (near) real-time measurements with EC Operator. In addition, it is equipped with an automated flexibility service providing/consuming (trading) agent (FSxA), which controls indirectly or directly (though xEMS, see ST1.4, Section 3.3.3) electrical loads and is able to place bids (FlexOffers) on the EC peer's behalf based on specified parameters (i.e., *bidding strategy* – bid prices, risk appetite, etc.).

Based on these assumptions, we split the EC Peer actor into two logical actors – Human EC Peer and EC Peer FSxA, and identify key information objects, associated to each in Figure 26.

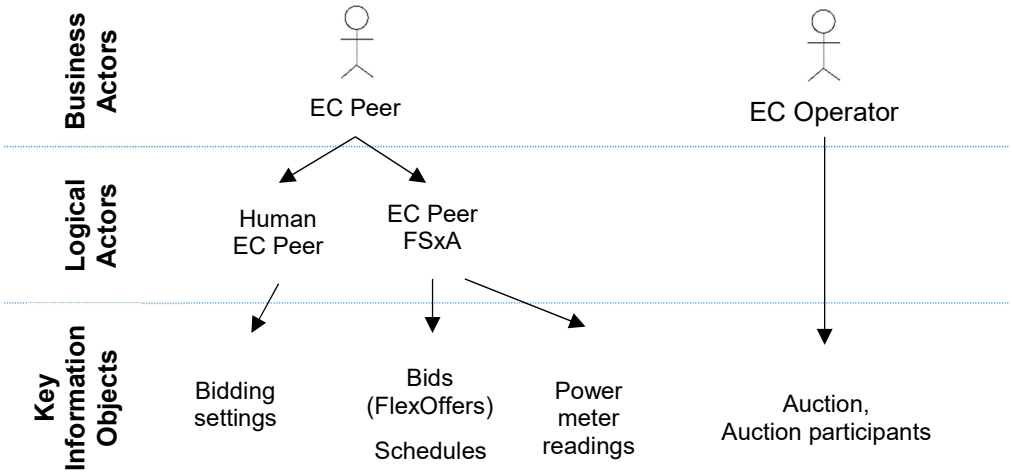

Figure 26 P2P marketplace actors and information objects

In the next section, we specify the most important of these, along with the related information objects.

7.4.2 Data Model

We will start with defining core information objects for use by the FlexTrading DAPP. As seen in Figure 27, FlexTrading DAPP builds up on a number of information objects from others DAPPs (Community Management DAPP and FlexCoin DAPP) and introduces some new ones. The most essential of those are discussed next.

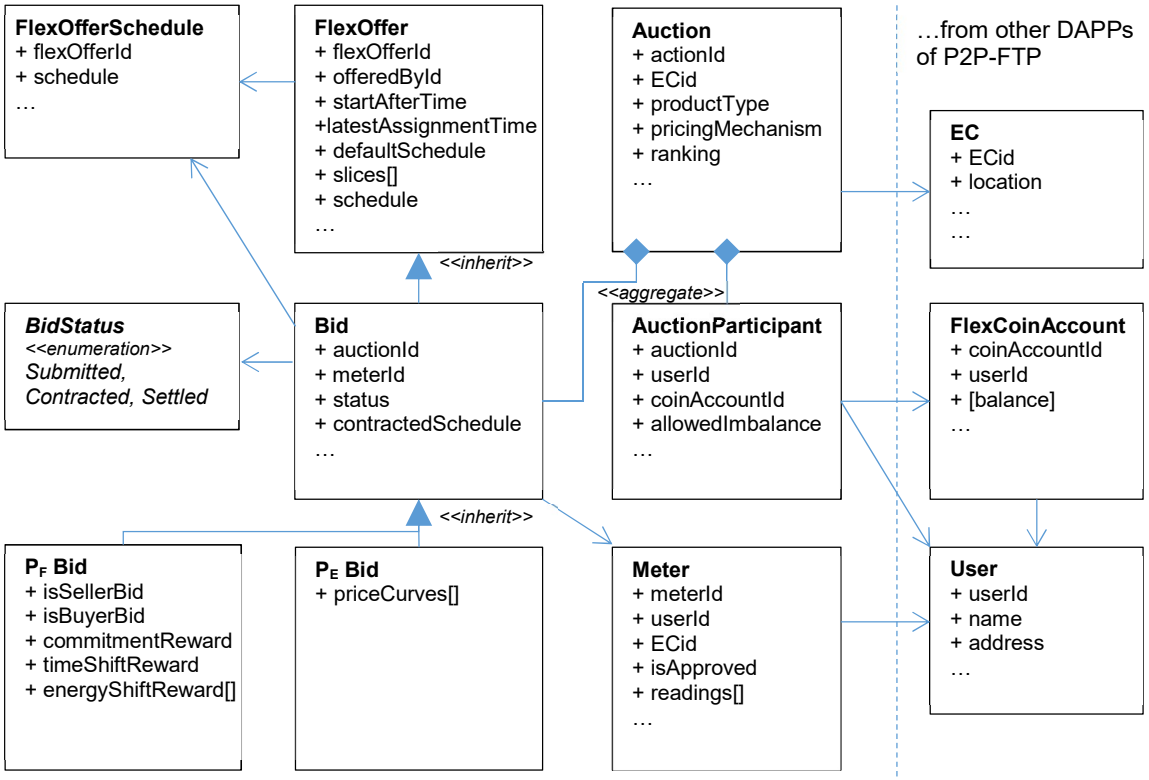

Figure 27 Simplified data model of the FlexTrading DAPP

- **Auction** – this class specifies an instance of a marketplace. It includes a unique Id of the marketplace (*actionId*), an Id of energy community it is linked with (*ECid*), a type of product (P_E

or P_F) to be traded (*productType*), pricing and ranking mechanisms (Section 7.3.3) to be used (*pricingMechanism*, *ranking*).

- **AuctionParticipant -** this class specifies a participant, which has (was granted by the EC Operator) access to a specific marketplace instance. The participant is linked with a particular P2P-FTP user (via *userId*) and its FlexCoint account (via *coinAccountId*). Specific parameters of the participant, which were agreed with the LEC operator (e.g., *allowedImbalance*), are also represented by this class.
- **FlexOffer** – this class specifies the EC peer's energy need/offer and associated flexibility, while covering a number of consecutive time intervals. The flexibility is defined by a number of constraints, including those defined by FlexOffer slices (see Section 7.1). The full specification of the FlexOffer attributes can be found in the WP2 deliverable D2.3.
- **FlexOfferSchedule** – this class specifies concrete energy amounts to be consumed and/or produced by the EC peer. The schedule is always linked with an instance of a FO (*flexOfferId*). Concrete energy amounts (*schedule*) are defined for each slice of the FO and they respect all FO constraints.
- **Meter** – this class specifies an energy meter of an EC peer. It can be a certified smart meter, or a sub-meter. In the latter case, it needs to be manually approved by the LEC operation to be able to use it for the P2P trading.
- **Bid** – this class extends the original FlexOffer class and specifies a bid targeting a particular market instance in the generic form. The bid and has several additional attributes, not available in the FlexOffer class. For example, *auctionId* links a bid with a particular marketplace instance; *status* specified the state of the bid (submitted, contracted, settled); *contractedSchedule* specifies concrete energy amounts to be consumed/produced by the participant. Two specializations of this class are possible: P_E Bid and P_F Bid.
- **P_E Bid** – this class specifies an offer or a request of the P_E product. It further extends the Bid with price curves (priceCurves, see Section 7.2) for each slice of a FlexOffer.
- **P_F Bid** – this class specifies an offer or a request of the P_F product. It further extends the Bid with a number of attributes relevant to P_F: whether the bid expresses an offer or a request of P_F, i.e., a flexibility service (*isSellerBid*, *isBuyerBid*); reward attributes of the offered/requested (*commitmentReward, timeShiftReward, energyShiftReward*).

7.4.3 Interaction between core components

The smart-contract of the FlexTrading DAPP will support a number of methods (transactions) which will manipulate these information objects on-chain – see the table below.

Table 9. Essential methods (transactions) of the FlexTrading DAPP smart-contract

Method Name	Method description
createAuction()	Creates a new instance of an auction
createParticipant()	Grants a new participant an access to an auction instance.
createMeter()	Registers a new power meter of an EC peer
placeBid()	Places a bid onto a specific instance of an auction. The method immediately returns a preliminary schedule, which includes prescribed energy amounts and preliminary prices.
updateAuction()	Reschedules all submitted (not yet contracted) bids. Clears the auction by bringing all pending FlexOffers (FOs, i.e., those which latest assignment time is due) into the contracted state (see Section 7.3.2).
reportMeasurements()	Upload EC peer's power meter measurements on chain. This is followed by the automatic validation of peer's contracted bids and settlement, during which specific amounts of FlexCoins is transferred between the

	EC-Peer's and the EC-Operator's FlexCoin accounts.

Before P2P trading can take place, the EC-Operator needs to create one (or more) auctions (*createAuction*), register its participants (createParticipant) as well as their power meters (createMeter). For (an) already created auction(-s), the EC-Operator is able to edit (or delete) auctions, participants, meters (methods for that are not shown in the table). After the configuration is complete, P2P trading can take place in an automatic fashion. A simplified interaction between the FSxA of EC-Peer and the FlexTrading and FlexCoin DAPP smart-contracts during this automatic process is shown in Figure 28.

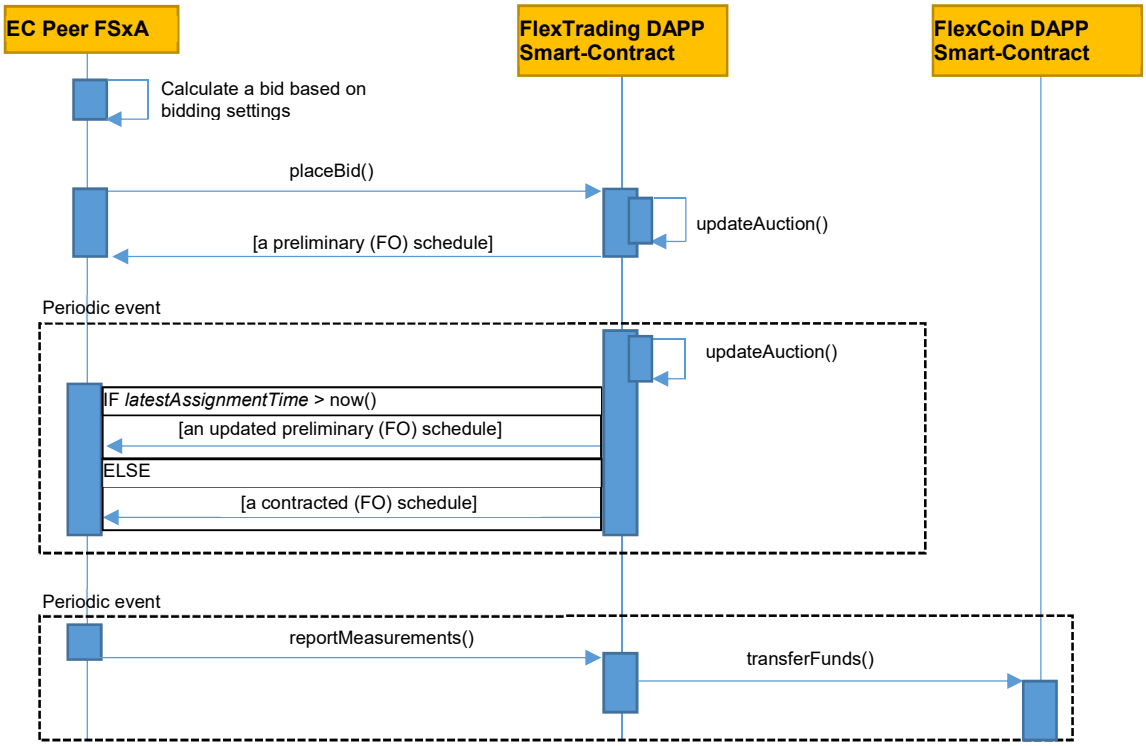

Figure 28 Simplified interaction between the FlexTrading DAPP FSxA and the smart-contract

Initially, the FSxA constructs a bid with all required FlexOffer parameters based on the bidding settings set my Human EC-Peer. The bid is then placed on the blockchain and handed by the FlexTrading DAPP smart contract. The smart contract automatically (and deterministically) matches this bid with the other bids within the auction and returns a preliminary schedule with indicative start time, energy amounts, and prices. Periodically, all auction bids are rescheduled. As a result, some bids receive preliminary schedules, the rest – contracted schedules with the final start time, energy amounts and prices. Finally, FSxA periodically reports power measurements to validate the execution of the contracted bids. Successfully validated bids, or bids without measurements available (in due time), trigger an automatic transfer of FlexCoin amounts from the EC-Peer to EC-Operator or vice versa.

7.4.4 Topology or Hyperledger Fabric network

The smart-contract presented in the earlier section (together with other relevant components) will be deployed on the HLF network. An example minimalistic deployment topology is shown in Figure 29.

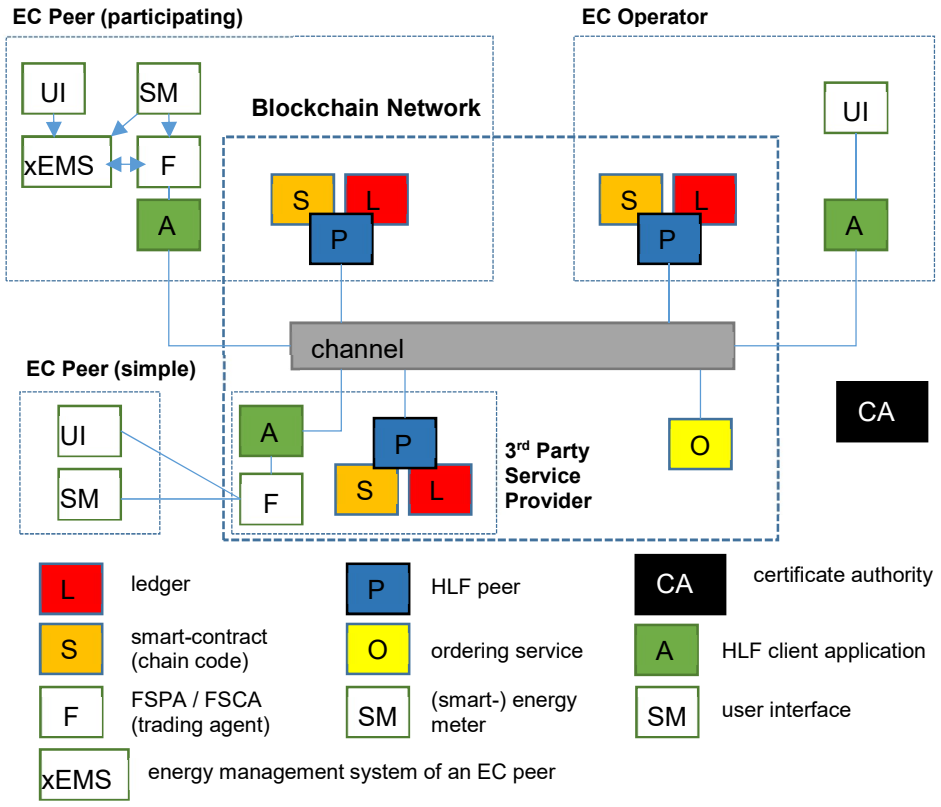

Figure 29 Example topology of the HLF network of the FlexTrading DAPP

Initially, all actors taking part in P2P trading within a single EC form a *consortium* – which may include EC peers, EC Operator, and some 3rd party organization like a local DSO (e.g., SWW) or a software service provider (e.g., CERTH, FlexShape, ICOM, INEA). For P2P trading in a specific EC, the members of the consortium are connected to the same common HLF *channel* ("channel" in the figure, see Section 4.2.3), which acts as a main communication medium allowing them to communicate to each other. The channel should be configured in such a way that only the EC-Operator can add new members to it. However, a less centralized channel configuration where a pre-defined share of all members has to agree in order to add new market participants is also possible.

As seen in the figure, the network includes a number of HLF network (*endorsing*) peers ("P" nodes), managed by different consortium members. For example, the EC Operator runs a HLF peer, which manages a copy of the FlexTrading DAPP's smart-contract ("S" node) and the ledger ("L" node). For resilience and availability, the EC Operator may run more HLF peers in other deployment configurations. In addition, the EC Operator runs a HLF *client application ("A" node)*, which (1) offers a user-interface ("UI" node) to a (human) user for managing the marketplace and is able to (2) send transaction proposals onto the channel by taking advantage of a Software Development Kit (SDK) for the Hyperledger Fabric. Similarly, two kinds of EC Peers are envisioned:

1. **EC Peer (participating)** - pro-actively participates in the HLF network. In runs a *HLF peer ("L" node)*, which manages a copy of the FlexTrading DAPP smart-contract and the ledger ("S" and "L" nodes). In addition, the participating peer has an *energy management system* ("xEMS" node) with an integrated (or an external) FSxA ("F" node) which acts as a *trading agent* towards the smart-contract with ability to send transaction proposals via a HLF client application ("A" node). The smart-meter ("SM" node) measurements are accessed by both the energy management system as well as the FSxA for load monitoring and bid validation purposes (ST1.4-ST1.7). In real-world setups, the HLF client application, FSxA, and xEMS ("A", "F", and "xEMS" nodes) can be integrated into a single software system offered to the EC Peer.
2. **EC Peer (simple)** – participates in the HLF network through a selected 3rd party service provider (e.g., FlexShape). The 3rd party service provider hosts its own instances of FSxA and the HLF

client application, as well as the HLF peer, and offers the EC peer an access to the HLF network though its own 3rd-party software platform, which integrates all these components. In this setup, EC peer can use a user interface ("UI" node) offered by the 3rd party platform, but needs to provide an access to (the data of) a power meter (and a remotely controlled relay/device, e.g., a smart-plug).

These exists a Certificate authority ("CA" node). This is an institution that issue *certificates* to consortium members and network nodes. These certificates define *user identities* and are used to map these identities to organizations, e.g., EC Operator. It is reasonable to assume that these exists a single common *Certificate Authority* that issues certificates to all – EC Operator, EC Peers, and 3rd party service providers. A blockchain network-wide external Certificate Authority might be additionally be used, if needed.

Finally, as explained in Section 4.2.2, the *ordering service* ("O" node) is responsible for ordering and packaging endorsed transactions into blocks - thus providing delivery guarantees. In production, this service should be provided by a cluster of nodes.

In order to transact, client applications ("A" nodes) submit transaction proposals to HLF (endorsing) peers and need to receive a predefined number of positive responses (i.e., endorsements) before they can send transactions to the ordering service. This number is fixed (in the endorsement policy) and embedded in the channel policy - defined at the moment of chain code instantiation. In order to maintain an efficient and scalable system, we recommend a policy where the signatures are required from: (1) a member peer associated with the EC Operator and (2) two or three other peers, belonging to different organizations.

7.4.5 Private bids on the ledgers

In this section, we discuss a distribution of actual data stored in the ledgers of different peers. As discussed earlier (Section 4.2.3), each HLF peer manages a *ledger* which stores all *transactions* from a particular blockchain *channel*. These interlinked transactions determine the *world state*, which is stored a *classical database* for increased speed and easy queries. However, in our P2P market design (Section 7.3), a bid (FlexOffer) submitted by some EC peer should remain *sealed* from the rest and shared only with the EC Operator. This can be achieved in Hyperledger Fabric using so-called *private transactions,* which were introduced in Hyperledger Fabric version 1.2. With this feature, data considered private can be shared with only authorized organizations (e.g., of EC Operator). Most importantly, this feature keeps private data confidential from an ordering service, which may be controlled by an organization unauthorized to see the data. In FlexTrading DAPP, a bid (FlexOffer) is split into two transactions: (1) a *private transaction*, holding its full information (attributes); and (2) a *public transaction* containing only basic attributes (which do not have to be protected) and visible for all market participants. The full details of the bid are stored in a private state database and shared between the invoking EC peer and the EC Operator. A hash of the transaction is stored on the public blockchain for auditing purposes.

This means that, in addition to the blockchain and the database storing the world state, every ledger of the participating peers additionally stores the (private) details of bids (FlexOffers) in a private state database – see Figure 30. The EC Operator peer(-s) thus maintains a number of such private state databases, each for a different participating HLF peer in the HLF network. In Figure 30, exemplifies the content of the ledgers of EC peer 1, EC peer 2, and the EC Operator.

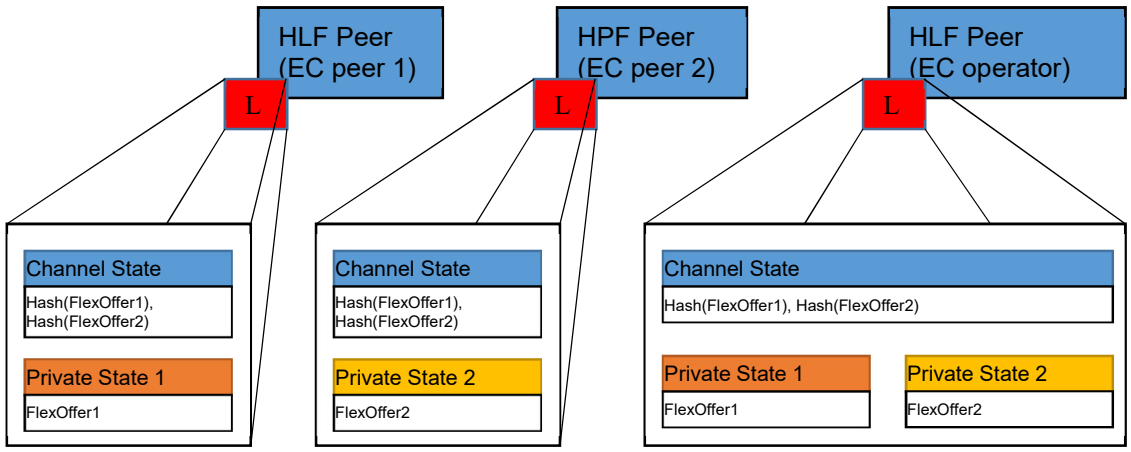

Figure 30 Ledger (L) contents of HLF peer nodes that store private data

7.5 Nested P2P Trading

The presented P2P marketplace design is primarily suited for P_E and P_F trading within a single EC. In this section, we discuss how this P2P marketplace can be used to cater a more advanced multi-EC trading use-case US5 from Section 2.5.

Figure 31 shows a simplified model of such *nested trading*. Here, EC1 is an example local energy community, where peers trade P_E and P_F products with each other. Actor A_1 takes the role of EC1 operator and EC2 peer at the same time. Therefore, A_1 on the behalf of EC1 has a capability to offer/request energy (P_E) or flexibility service (P_F) products to/from the outer EC2. To trade on EC2, A_1 continuously submits P_E bids for energy and/or P_F bids for flexibility services onto the EC2 market (bidding area). On successful trades on the EC2 market, A_1 includes counter P_E and/or P_F bids into the EC1 market (bidding area).

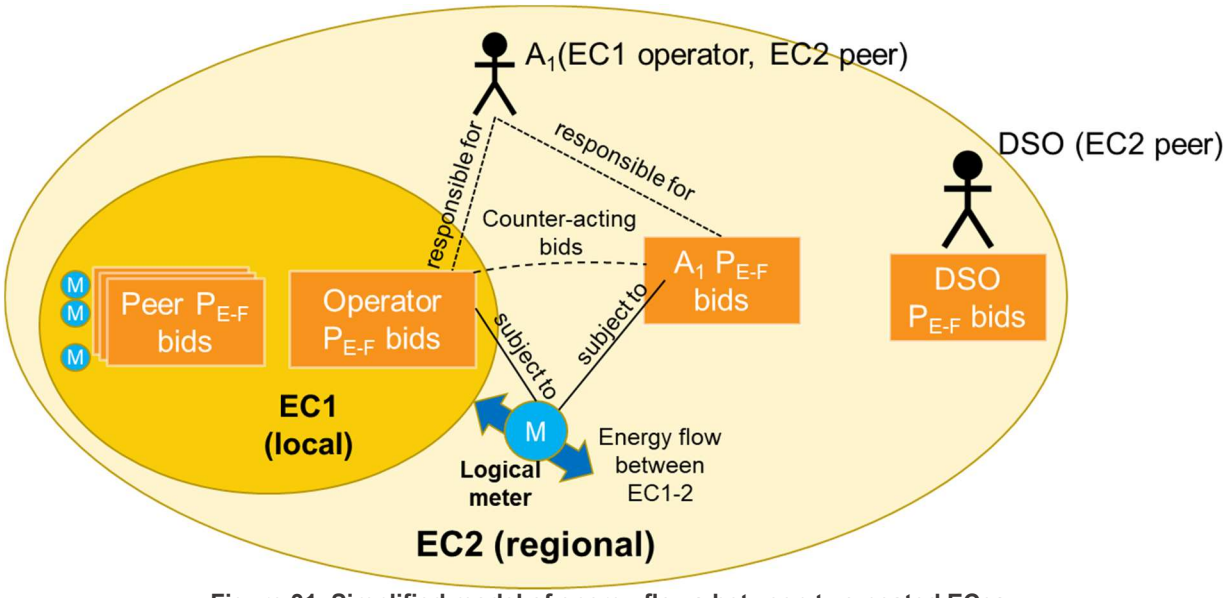

Figure 31. Simplified model of energy flows between two nested ECes

8 Specification of API for Integration Services

This section focuses on Requirement 5 of P2P-FTP (R5, Section 3.1.5) and provide the specification of the APIs on the P2P-FTP-side, that are exposed to other related FEVER systems, primarily FTP (WP2).

To cater scenarios with a trusted market operator, FTP from WP2 needs to have access to the Community Management DAPP and FlexCoin DAPP (see US3, Section 2.3). In the tables below, we specify such an "outside-facing" REST API, which is exposed by P2P-FTP to FTP (as well as other FEVER systems). This APIs includes only the essential end-points, required for integrating P2P-FTP with other FEVER systems. Actual functionality behind this API is offered by Community Management and FlexCoin DAPPs (see Sections 5-0).

Table 10. API for integration

Endpoint	Parameters	Return	Description	Trace to requirements
Community Management DAPP API				
/community/login	• User credentials • User certificate [optional]	Authentication Token	Login operation provides FTP users to link and use their P2P-FTP user account, providing access to P2P-FTP functionality	SC 1.3
/community/user/ GET	• Authentication or ID Token for EC-Operator	User Identifiers in EC	Allows EC-Operators to view users of a given EC	SC 1.2
/community/user/{user_id} GET	• Authentication or ID Token for EC-Operator User Identifier (user_id)	User Data	Allows EC-Operators to view user data for given user in EC	SC 1.2
/community/user POST (optional)	• Authentication or ID Token for EC-Operator User Data	User Identifier	Allows EC-Operators to create new users for given EC	SC 1.2
/community/user/{user_id} PUT (optional)	• Authentication Token or ID for EC-Operator • User Identifier (user_id) • User Data (updated)	User Identifier	Allows EC-Operators to edit users (such as, changing user roles) for given EC	SC 1.2

FlexCoin DAPP API				
/flexcoin/balance	• Authentication or ID Token	FlexCoin Balance	Allows Peers and EC-Operators to view their FlexCoin account balance Utilizes **/account/{accId}** from FlexCoin HTTP API	SF 1.1, SF 1.8 (for EC-Operator)
/flexcoin/log	• Authentication or ID Token • Date Range	FlexCoin Transaction log	Allows Peers and EC-Operators to view their FlexCoin transactions for a given date range Utilizes **/account/log/{accld}** from FlexCoin HTTP API	SF 1.1, SF 1.8 (for EC-Operator)
/flexcoin/claimdeposit (optional)	• Authentication or ID Token • FlexCoin Amount • Claim of Contribution		Allows Peers and EC-Operators to request a deposit of FlexCoins into their FlexCoin account based on a claim of contribution Utilizes **/mint** from FlexCoin HTTP API	SF 1.2.1, SF 1.8 (for EC-Operator)
/flexcoin/deposit (optional)	• Authentication or ID Token • FlexCoin Amount • Receipt of E-banking transaction		Allows Peers and EC-Operators to request a deposit of FlexCoins into their FlexCoin account based on a banking transaction Utilizes **/mint** from FlexCoin HTTP API	SF 1.2.2, SF 1.8 (for EC-Operator)
/flexcoin/withdraw (optional)	• Authentication or ID Token • FlexCoin Amount		Allows Peers and EC-Operators to request withdrawal of FlexCoins from their FlexCoin	SF 1.3, SF 1.8 (for EC-Operator)

		account. Utilizes **/burn** from FlexCoin HTTP API	
/flexcoin/transfer	• Authentication or ID Token • FlexCoin Amount • Recipient User Identifier	Allows Peers and EC-Operators to send FlexCoins from their account into another P2P-FTP users FlexCoin account Utilizes **/account/transfer** from FlexCoin HTTP API	SF 1.4, SF 1.8 (for EC-Operator)
/flexcoin/proxytransfer (optional)	• Authentication or ID Token • FlexCoin Amount • Sender User Identifier • Recipient User Identifier	Allows Peers and EC-Operators to send FlexCoins from an account of a 3rd party into another P2P-FTP users FlexCoin account. The 3rd party has to give prior approval to such transfers Utilizes **/account/transfer** from FlexCoin HTTP API	SF 1.4, SF 1.8 (for EC-Operator)
/flexcoin/hold (optional)	• Authentication or ID Token • FlexCoin Amount • Recipient User Identifier	Allows Peers and EC-Operators to hold FlexCoins in their account Utilizes **/account/hold** from FlexCoin HTTP API	SF 1.4, SF 1.8 (for EC-Operator)

Table 11. API data object descriptions

Data object name	Data object description
User Credentials	Data used to uniquely authenticate a user in P2P-FTP. Includes, but not limited to: account user name (or registered email address), account password.
Authentication or ID Token	Generated API token during successful user authentication during login that is uniquely

	associated with the user session.
User Identifier	Unique identifier for given user in P2P-FTP.
User Data	User data, including: user name, user role, user creation date
Date Range	Two date objects: beginning of date range and end of date range.
FlexCoin Balance	Data for given user's FlexCoin balance, including, but not limited to: balance of currently disposable FlexCoins, FlexCoin transactions pending approval (e.g. FlexCoin amount for claim of contribution pending EC-Operator approval)
FlexCoin Transaction Log	List of FlexCoin transactions within a time period. Each transaction record includes: Sender User Identifier, Recipient User Identifier, FlexCoin amount, Timestamp, Reason for Transaction (optional)
FlexCoin Amount	Number of FlexCoins attributed to a given transaction
Claim of Contribution	Claim for a contribution made to the EC for which the contributor expects to be renumerated a specific amount of FlexCoins
Receipt of E-banking transaction	Document or transaction data from an E-banking system, confirming that a specified amount of money has been transferred to a bank account

9 Cloud Facility and CI/CD pipeline specification

This section focuses primarily on requirement 7 of P2P-FTP (R7, Section 3.1.7) and specifies the P2P-FTP deployment environment. The proposed deployment environment, as well as utilized system development, monitoring, and deployment tools are described in the following subsections.

9.1 Cloud Facility Infrastructure

9.1.1 Hosting Environment

Modern containerization solutions provide an out of the box solution by natively supporting deployment of the different HLF service components (Peers, Orderers, etc.) in an isolated and distributed fashion.

Following this paradigm, each HLF component service instance will be run in its own container. Containers utilize a virtualization mechanism at the operating system level, consisting of one or more processes running in an isolated sandbox environment. From the perspective of the processes, they have their own port namespace and root filesystem. The resources allocated to each container (memory and CPU) are fully configurable and some container implementations also allow for I/O rate and resource limiting. Popular examples of container runtime technologies include Docker [14] and Containerd [15].

Once a service is packaged as a container image, it can then be launched as a standalone virtual machine (VM). It is possible to run multiple containers on each physical or virtual host. However, keeping track and administrating of containers becomes increasingly complex as more containers are added to the host system.

A HLF network is agnostic to the methods used to deploy and manage it. HLF components can therefore be either hosted in a distributed manner over different geographical location or can be collocated in the same host system.

For the purposes of this project all HLF components will be hosted on premise and in a private-cloud solution implementing Kubernetes [16] based cluster solution.

9.1.2 Kubernetes based orchestration

A cluster manager such as Kubernetes can facilitate management of service containers, by treating a host as a pool of resources. It decides where to place each container based on the resources required by the container and resources available on each host. It can also provide administration tools and dynamically manage workloads including moving instances from one virtual host to another based on load and dynamically allocate resources such as CPU and memory.

Kubernetes features a number of helpful tools that have made it a popular container management platform for deploying and managing Fabric networks. Some of those services offered are:

- High Availability
- Self-Healing and resiliency to failure
- Dynamic scalability
- Resource monitoring
- Centralized container logging
- Role based access control

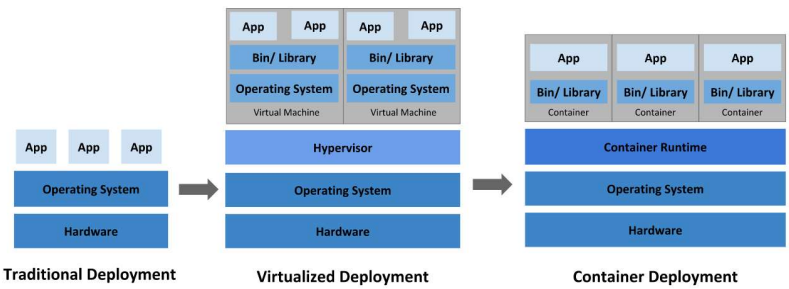

Figure 32 Application Deployment Methods [17]

An advantage of Kubernetes, which meets the requirement R7, is the ability to define services and deployments through configuration files which can be later applied on multiple, completely separate clusters.

9.1.3 Resource allocation

On key consideration when designing a HLF network is ensuring that the hosting environment possesses enough resources for the components to run effectively. The sizing of the resources needed (CPU, RAM, Disk, Network I/O) will largely depend on the specific use case requirements. Generally, when designing a HLF network the following empirical rules apply:

Joining a peer to several high-volume channels, results in more CPU and memory than joining it to a single channel. As a rough estimate, it is recommended to allocate three times more resources for a peer as for a single ordering node (it is recommended to deploy at least three and optimally five nodes in an ordering service).

Similarly, the resources needed for a CA as approximately a tenth of what is needed for a peer.

Additionally, adequate persistent storage will have to be provisioned. The use of persistent storage ensures that data such as MSPs, ledgers and installed chaincodes are not stored on the container filesystem, preventing them from being destroyed if the containers are destroyed.

Although the above facilitates the calculation of a rough estimate of the required resources, the only proven way to accurately estimate the actual resource requirements of a production grade network is by deploying a proof-of-concept network at scale and test it under load.

Following the above reasoning the following resource have to be allocated for hosting of an initial POC network.

Table 12 Infrastructure Resource Alocation

	UC1 - Single EC (intra - EC trading)	UC2 - 2 ECs (intra - EC trading)
Organizations	EC1 + Ordering Org	EC1 + EC2 + Ordering Org
Users	20 EC End Users	40 EC End Users
Ordering Service (Raft)	3 Nodes	5 Nodes
Organization Peers	4 (2 Endorcing, 1 Committing, 1 Leader)	8 (4 Endorcing, 2 Committing, 2 Leaders)
Channels	1	2
Certificate Authorities	2 (1 EC + 1 Ordering Org)	3 (2 EC + 1 Ordering Org)
CPU	4 Vcores	8 Vcores
RAM	8GB	16GB
Storage	50GB (Containers + Data Storage Requirements)	75GB (Containers + Data Storage Requirements)

9.1.4 Resource Monitoring

Monitoring consumed resources and API transactions of an HLF network, can provide an early warning system for application deterioration or failure, a 'big picture' analysis of the health of each component, as well as an indicator as to where and when, so potential issues can be identified and resolved quickly.

In the same way that the interaction of different components within a HLF network happens over agreed upon protocols and data formats, a standardized way has to be devised on how to log health and

diagnostic events that will ultimately end up in an event store for querying and viewing. Different stakeholders need to agree on a single logging format as there needs to be a consistent approach to viewing diagnostic events in the application as a whole.

To efficiently scale a HLF network and provide service reliability, it is needed to understand how the application behaves in terms of resource usage when it is deployed. Application performance in a Kubernetes cluster can be examined by monitoring the containers, pods, services, and the characteristics of the overall cluster. Kubernetes provides detailed information about an individual components resource usage at each of these levels. This information allows to monitor the real-time application performance and in order to evaluate where bottlenecks occur and remove them to improve overall performance.

In Kubernetes, application monitoring does not depend on a single monitoring solution. Different resource metrics or full metrics pipelines can be used to collect monitoring statistics.

Kubernetes Dashboard

Kubernetes Dashboard is a web-based Kubernetes user interface. It can be used to deploy, troubleshoot and manage a Kubernetes cluster resources. The solution provides an overview of the applications running on your cluster, as well as for creating or modifying individual Kubernetes resources. Dashboard also provides information on the state of Kubernetes resources in your cluster and on any errors that may have occurred.

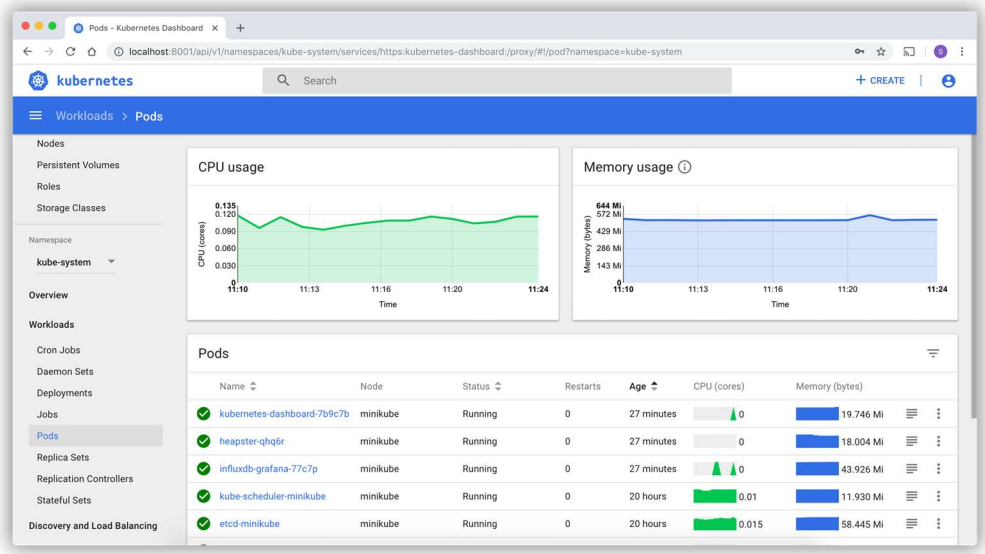

Figure 33 Kubernetes Web Dashboard [18]

Prometheus

Prometheus [19] monitoring is an entire monitoring stack around Prometheus timeseries database server that collects and stores monitoring metrics. This solution also includes Grafana [20] for visualization of the results in the form of dashboards, and often a number of exporters that are usually comprised of small sidecar containers that transform service specific metrics into Prometheus metrics format. It records real-time metrics in a time series database built using a HTTP pull model, with flexible queries and real-time alerting.

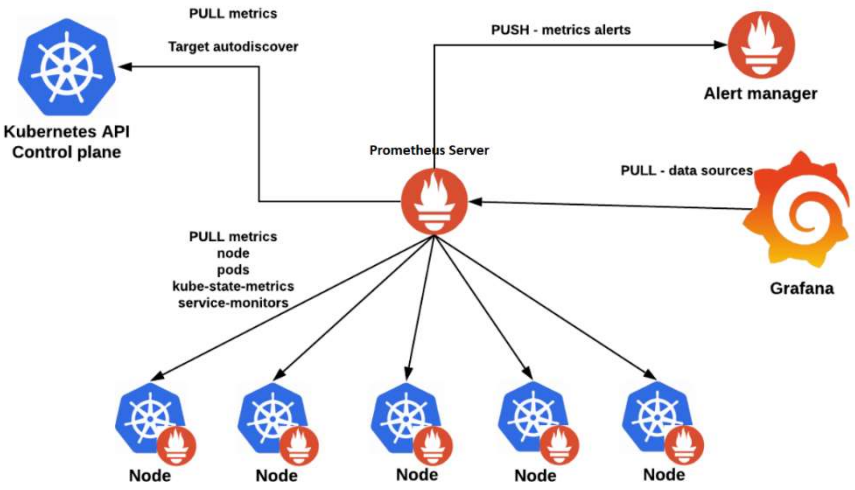

Figure 34 Kubernetes-Prometheus-Grafana setup representation [21]

9.1.5 Centralized Logging

A HLF network architecture involves discrete, loosely coupled components that work within their own process boundaries. Thus, logging is a perfect example of a cross-cutting concern, that needs to span over numerous modules at different levels of the code base. Consequently, when HLF application is split into isolated service containers, logging is also split over every service component.

A single logging solution thus provides a common space where logs can be collected, regardless of the service from which the log message originated initially. Additionally, cases exist where errors may span over multiple service instances. This introduces the need for a centralized logging solution that has to be able to interface with different technologies and aggregate collected log data from each individual component in order to be analysed.

Another aspect to consider is the ability to efficiently query and visualize consolidated log data. The application should offer an easy-to-use interface, where data can be queried and visualized in order to extract key metrics insights from the deployed services.

Cluster-level logging architectures require a separate backend to store, analyse, and query logs. Kubernetes does not provide a native storage solution for logging data. Instead, there are many logging solutions that integrate with Kubernetes.

Elasticsearch – FluentD – Kibana (EFK)

Elasticsearch [22] is a real-time, distributed, and scalable search engine which allows for full-text and structured search, as well as analytics. It is commonly used to index and search through large volumes of log data, but can also be used to search many different kinds of documents.

Kibana [23] and Elasticsearch is commonly deployed alongside Kibana, a powerful data visualization frontend and dashboard for Elasticsearch. Kibana allows you to explore your Elasticsearch log data through a web interface, and build dashboards and queries to quickly answer questions and gain insights into the applications deployed on Kubernetes.

Fluentd [24] is used to collect, transform, and ship log data from each container to the Elasticsearch backend. Fluentd is a popular open-source data collector that is set up on each Kubernetes node to tail container log files, filter and transform the log data, and deliver it to the Elasticsearch cluster, where it will be indexed and stored.

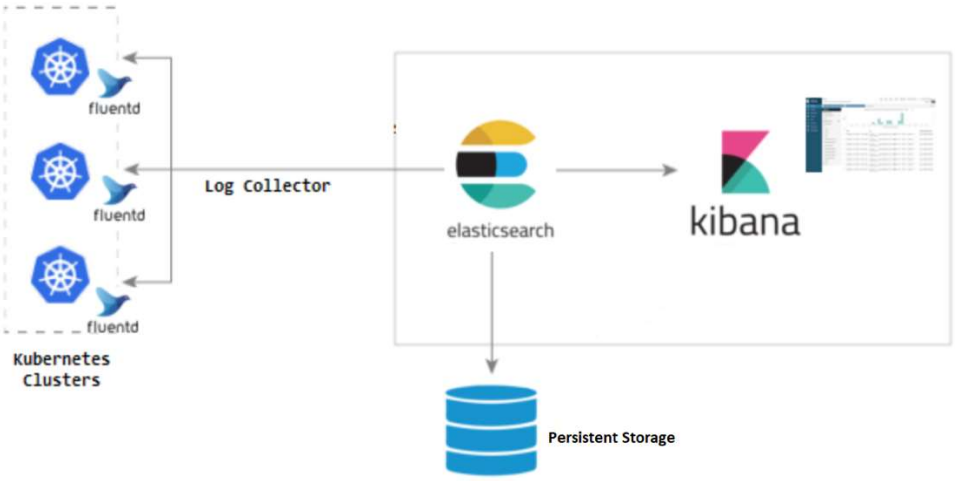

Figure 35 Kubernetes – Fluentd – Elastic - Kibana integration [21]

9.1.6 Dynamic scalability and self-healing

The service load of a deployed HLF network may vary depending on time of the day, as there may be cases where existing component instances may not be able to support the required workloads at peak times.

Dynamic scalability ensures that the application can automatically scale-up by automatically creating new service instances as they are needed. The load is automatically redistributed to fresh instances upon their creation via load balancing. Likewise, where the application load is low, the application can automatically scale down, by removing service instances that are idle and redistribute network traffic accordingly to the remaining instances.

This ensures that available system resources (CPU, memory, etc.) are efficiently managed, as maintaining service instances that are idle consumes resources making them unavailable to other services that may need them and subsequently decreasing efficiency.

For availability, the application also needs to be resilient to failures. In the case that an application stops responding or its performance significantly degrades, the platform should be able to spin up new instances (or restart already existing) automatically. The service should be able to manage service instances while ensuring a seamless transition and virtually zero downtime. An addition to these resiliency requirements, is that data shouldn't be lost, and data needs to remain consistent.

9.1.7 Role Based Access Control

All Kubernetes resources, such as pods and deployments, can be logically grouped into a namespace. Namespaces are intended for use in environments with many users spread across multiple teams, or projects and provide a way to logically divide a Kubernetes cluster and restrict access to create, view, or manage resources. Therefore, allowing only for specific users to only interact with resources within their assigned namespaces.

As already mentioned in 5.2 the Role-Based Access Control (RBAC) mechanism is one of the well-described access control mechanisms in the literature.

Kubernetes RBAC provides the capability to assign users, or groups of users, assign specific permissions to those users, like create or modify resources, or view logs from running application workloads. These permissions can be scoped to a single namespace, or granted across the entire cluster.

In order to achieve this, Kubernetes implements the concept of roles and role bindings that allow assigning permissions to users or resources both at a namespace and/or at a cluster level. This approach allows for the logical segregation a Kubernetes cluster, with users only being able to access

the application resources in their assigned namespace.

An RBAC *Role* object contains the rules that represent a set of permissions that are given to a user within a specific namespace. Permissions defined by a role are purely additive (i.e., there are no "deny" rules). Once user specific roles have been defined, *RoleBindings* can then be defined in order to assign those roles to a specific namespace. It is worth noting that cluster wide roles and role bindings also exist, identified as *ClusterRoles* and *ClusterRoleBindings*.

Admin users assigned to a namespace can create or modify users, or grant access privileges to users of that namespace. Only the cluster admin has full access to system namespaces and cluster-wide resources.

The diagram below presents how RBAC can be organized in a Kubernetes cluster.

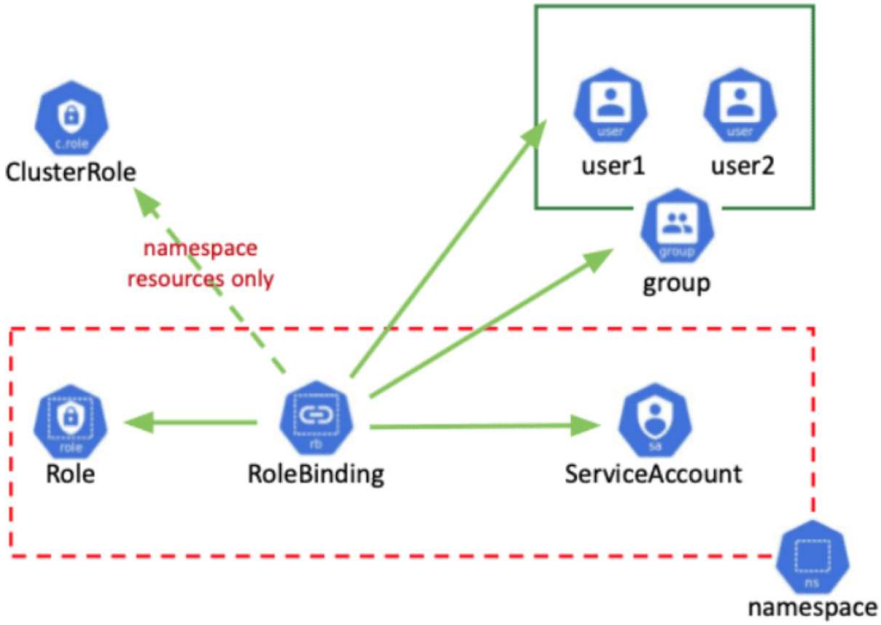

Figure 36 Kubernetes RBAC indicative rule enforcing [22]

9.2 HLF Infrastructure

A brief introduction on the underlying Blockchain technology – i.e., HLF – which realizes the distributed P2P-FTP is provided in section 4.2.

In the context of this section, there are 2 HLF network nodes of particular interest, namely: Peers and Orderers, as those have a direct participation in the chaincode lifecycle and the transaction flow.

At its core, HLF utilizes a set of *D*ocker Images, precompiled G*o*lang binaries and *c*ustom *c*onfiguration files, to properly define and build the desired Blockchain network infrastructure. Peer, CA, Orderer, fabric tools and other official images are based on the set of the provided base *D*ocker Images, which will be utilized to build the HLF network components.

Fever HLF Network Architects (system admins) define and setup a new HLF Network, based on the access rights defined by the Kubernetes cluster RBAC, as indicated by requirement R1 (subsection 3.1.1) . During the HLF Network setup phase, the network admin builds multiple containers (i.e., network nodes) utilizing the set of base images. The custom configuration files describe in detail a number of HLF network and channel level aspects, such as: the consortium members (organizations), the ordering service technology (e.g., Kafka, Raft) and configuration, the enforced policies and application specific configurations. Once the network is built, the precompiled binaries can be invoked through the HLF command line interface (CLI) in order to interact with specific organization nodes.

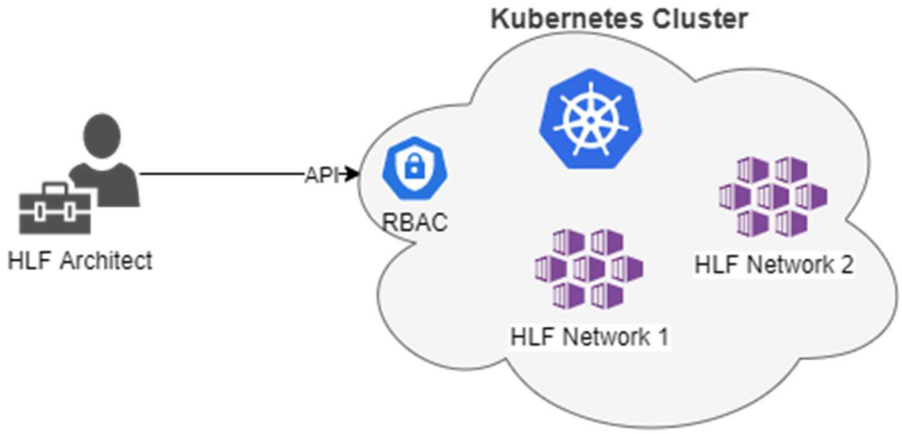

Figure 37 Kubernetes Cluster Access

Figure 37 illustrates how an HLF Network Architect may interact with the Kubernetes cluster.

9.2.1 Ordering Service

Hyperledger Fabric utilizes the Ordering Service to pack multiple transactions into blocks and distribute them to the channel's peers for validation and commitment to the immutable ledger.

The Ordering Service is only responsible for packaging transactions in strictly ordered blocks and distribute them to the channel's peers. It is not responsible to validate the requested transaction, neither has it the means to do so (no chaincode installed). To ensure transactions validity, before accepting and committing a block to the immutable ledger the channel's peers validate that each transaction in the block complies to the corresponding chaincode (or channel) endorsement policy, meaning all the required parties have endorsed the corresponding transaction in the block.

The Ordering Service is comprised of one or more ordering nodes, which are capable of handling multiple transaction proposals concurrently, reliably and in a deterministic manner. During the development and testing phases a single node ordering service may be utilized, although in staging and production environments an Ordering Service should employ multiple ordering nodes in order to meet high availability and reliability requirements.

In cases where the Ordering Service is comprised of multiple Ordering Nodes an implementation decision needs to be made in order to ensure the strictness of the transactions order among Ordering Nodes. Kafka, Solo and Raft are the supported implementations by HLF, with Raft being the official recommended for the latest LTS version (HLF v2).

Fever solution employs the Raft implementation. Raft is a crash fault tolerant (CFT) Ordering Service implementation, meaning the service will still operate properly even if a number of nodes are down, as long as the majority of the nodes are fully functional. The implementation relies on the Raft protocol [25], [26], which enables the CFT capabilities, in combination with etcd [27], a distributed key-value store. Among other advantages Raft ensures identical log (strict transaction order in HLF case) across ordering nodes.

9.2.2 Peers

In the context of HLF a Peer is a network node, while from the Kubernetes Cluster point of view it is a running pod, operating on behalf of a consortium member (Organization). Peers are very important entities of the HLF network as they are responsible of keeping a copy of the immutable ledger and the state database for each of the channels they participate in, endorse and validate transaction proposal as well as commit blocks to the ledger.

Peers are also required for the installation and/or the update of new chaincode versions. The infrastructure has to be configured, so that it is ensured that the defined peers will be present and

operational, requirement which can be fulfilled by Kubernetes.

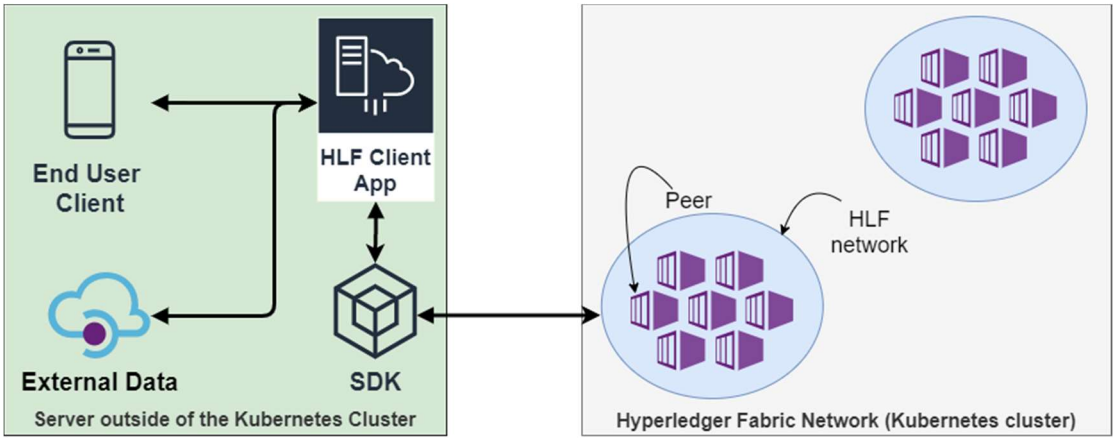

Figure 38 HLF Components Premises

FEVER DAPPs (frontend, backend - client application utilizing fabric SDK) are maintained outside of the Kubernetes Cluster where HLF Network lives. Organization admins can access their peers through the Fabric SDK in order to install, approve, commit and later on invoke chaincode on their peers, as shown in Figure 38. The majority of the organizations have to approve a chaincode upgrade before it can be utilized for transactions. This is a feature introduced in HLF version 2.x along with other features which provide more flexibility.

9.2.3 HLF Network Monitoring

On top of Kubernetes Cluster monitoring and logging an extra layer of HLF network specific monitoring needs to be setup. The main objective of this layer is to provide insight on the activities taking place inside the HLF network. Hyperledger Explorer [28] is an open source tool providing such insights for different levels of the network (node, block, transaction).

9.3 Version Control System

Version control is a system that enables its users (e.g., developers) to keep track of previous records and the intermediate changes applied to single file or a set of files over time by any number of users. *Version Control Systems (VCS)* provide a number of features such as merging multiple changes made across files automatically or semi-automatically, comparing and discarding unwanted changes, allowing users to create new branches and change files on them without affecting the main branch and many more. Git, SVN and Mercurial are a few examples of such version control systems.

There is a number of Git [29] managing tools, such as GitHub [30] and GitLab [31], which apart from git capabilities provide features such as: issue tracking, Continuous Integration and Deployment (CI/CD) as well as other tools to improve productivity and team collaboration.

For the development of FEVER solution, a private GitLab [32] git managing and DevOps tool is utilized. The solution is organized as a group of projects, each containing the required code for a separate part of the toolbox. Figure 39 provides an indicative group representation, in which 3 distinct projects are created, one for each DAPP.

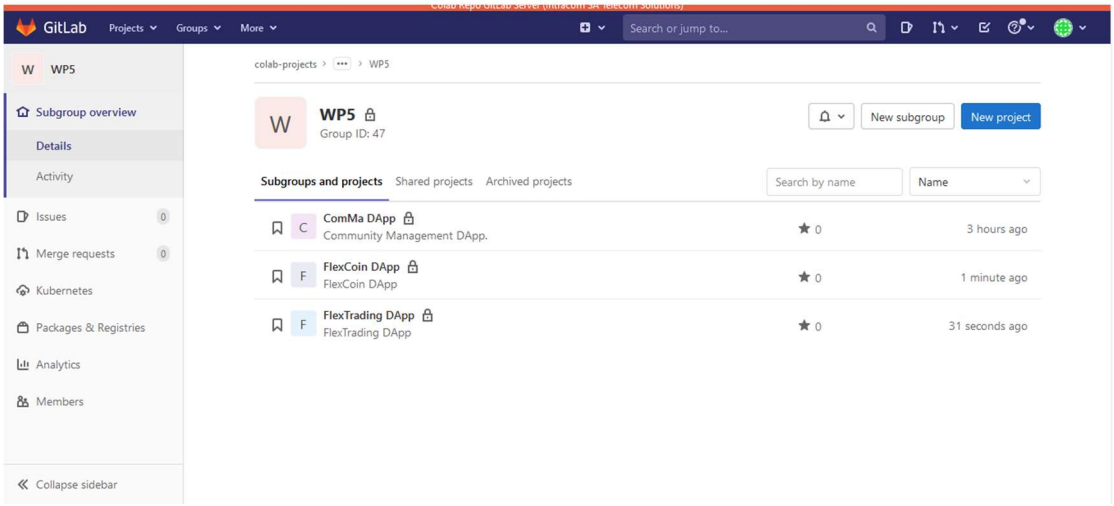

Figure 39 DApps VC Management System

9.4 Deployment Automation and CI/CD Pipelines

Deployment automation provides the ability to move software between testing and production environments by using automated processes, leading to repeatable and reliable deployments across the software delivery cycle. This enables the release of new features and applications quickly and efficiently, while removing the need for human intervention in application deployments.

Continuous Integration (CI) provides the tools to the development team to merge, build and test different code combinations in a frequent and clear manner. It ensures interoperability across builds and test setups while also reduces the overall required time.

Following the CI comes the *Continuous delivery (CD)* phase. CD automates the process of building, packaging and delivering all the introduced code changes to a testing, staging or production environment for manual deployment. The result of a successfully Continuous Delivery Job is a new release every time such a job is triggered and thus a team could eventually configure CD tools to provide a new release in a daily or weekly manner according to their needs.

Continuous deployment (CD) differentiates from Continuous Delivery by adding an extra step to its workflow. Once CI has finished building and testing new code successfully, Continuous Deployment releases the code to the production environment. The difference with Continuous Delivery is that it does not only release the new package but also deploys the code for the end user to see and make use of without any human intervention. Continuous Deployment may be configured to follow specific rules before each deployment takes place in order to ensure that only high-quality code makes it to the production.

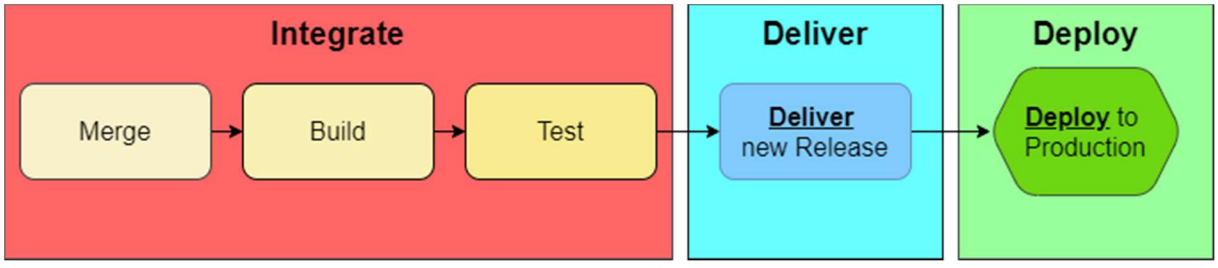

Figure 40 CI/CD Pipeline

As illustrated in Figure 40 a CI/CD Pipeline can be defined to run through the required logic to build, test, release and deploy the newly merged pieces of code.

FEVER solution employs CI/CD pipelines to take advantage of their added value. Developers will focus

on writing code as well as comprehensive tests for their code. The build, test, package release and deployment processes are handled by automated CI/CD pipelines, which will be defined and configured by each DAPP development team, with instructions provided in a gitlab-ci file. Such a file is provided in Figure 41, in which instructions are provided to build a Docker image based on the Dockerfile and the code in the project (repository).

```
docker-build:
  image: docker:latest
  stage: build
  services:
    - docker:dind
  before_script:
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
  script:
    - docker build --pull -t "$CI_REGISTRY_IMAGE" .
    - docker push "$CI_REGISTRY_IMAGE"
  rules:
    - if: $CI_COMMIT_BRANCH
      exists:
        - Dockerfile
```

Figure 41 Gitlab-ci configuration file

In order to complete those steps in a secure and reliable manner, an isolated machine needs to be configured with a Runner [33] installed on it. A Runner is an application capable of parsing instructions provided in CI/CD configuration files and run them as jobs in a pipeline. Jobs may run concurrently or sequentially. The result of a Pipeline's job may be an artifact, an archive of files and folders. A deployment stage may be defined in the pipeline to automatically deploy the generated artifact directly to the production environment.

Figure 42 shows a list of previously completed pipelines, along with pipeline specific information, such as: status, Pipeline ID, pipeline stages and overall duration. The capability is also given to download pipeline generated artifacts on demand.

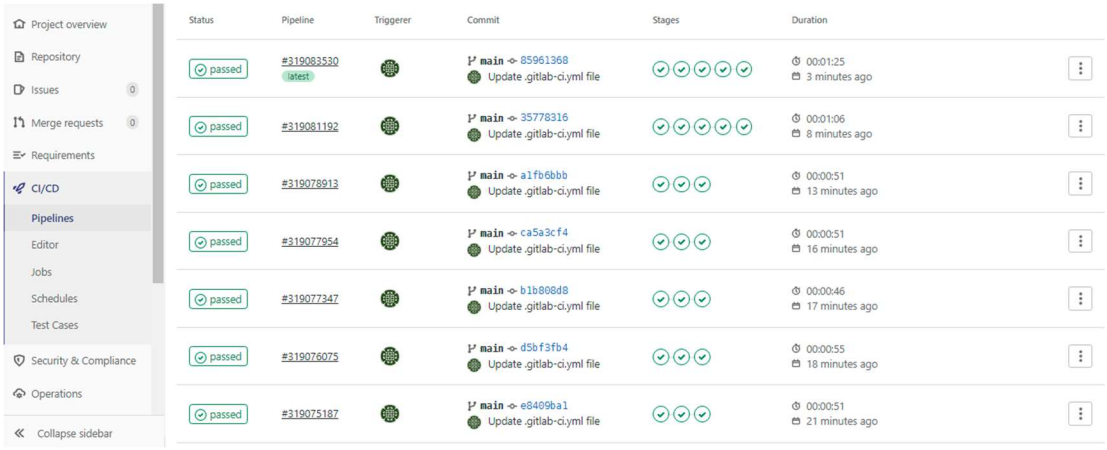

Figure 42 Pipelines List Overview

Dynamic visualization features are also supported as illustrated in Figure 43, which provides insight of the running, completed and failed jobs and their dependencies, while Figure 44 illustrates the job dependencies by stage in a colourful representation.

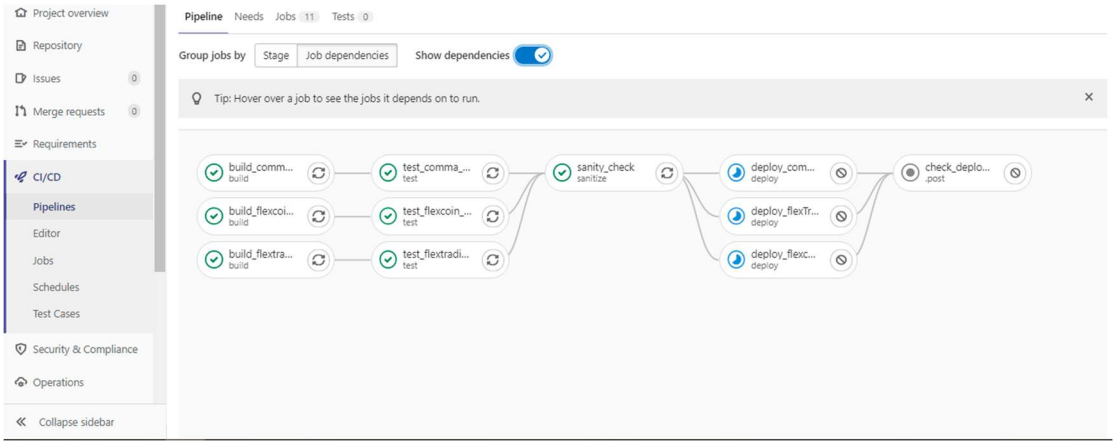

Figure 43 Pipeline jobs

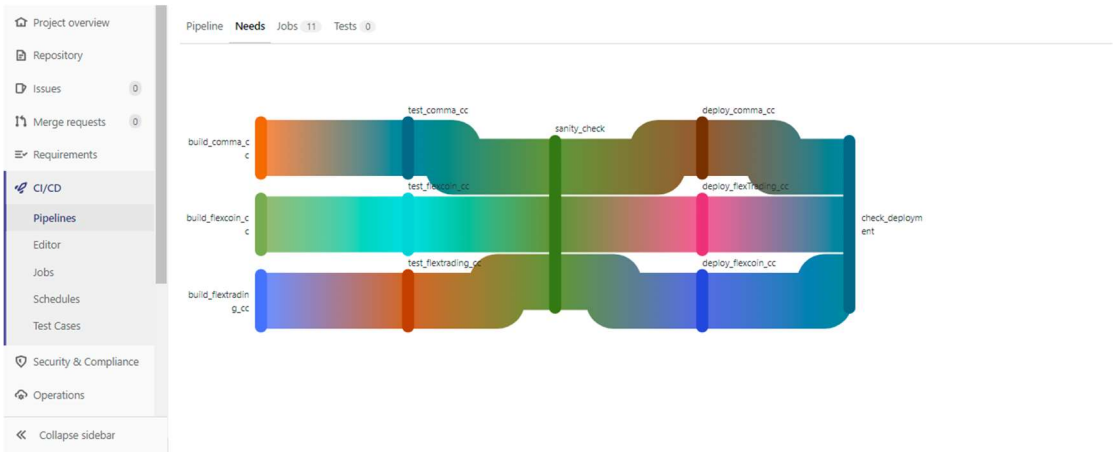

Figure 44 Pipeline depedencies Illustration

Specifically for the case of chaincode updates, CI/CD Pipelines can dramatically simplify the required process. By properly configuring a CI/CD Pipeline, chaincode developers can make changes to their code, commit the changes to the repository and let configured pipeline handle all the step from building and testing binaries, building new chaincode Docker images and publish them to the private image registry, which is utilized by the Kubernetes cluster to create pods. Following these steps, developers will only need to manually complete chaincode lifecycle steps.

10 Conclusion

This document summarizes the efforts of Tasks 5.1-5.3 (set in FEVER WP5) and specifies a so-called Peer-To-Peer Flexibility Trading Platform (P2P-FTP). P2P-FTP is a blockchain-based software platform for peer-to-peer flexibility trading (P2P-FTP), and it is an integral part of the Peer-to-Peer Flexibility Trading Toolbox of WP5.

P2P-FTP targets so-called *energy communities* (ECs) where active electricity consumers and/or producers (prosumers, peers) have a need to transact *energy*, *monetary*, and (other) *communal* assets in the peer-to-peer (P2P) fashion without a fully trusted marketplace operator. For such communities, P2P-FTP offers a novel concept of *FlexCoins,* which allows (1) quantifying each peer's contribution to the welfare of an EC, and (2) transferring such contributions between EC peers, being a mix of the currency and *bonus-point* mechanisms. Also, it offers an auction-based P2P marketplace solution, which allows electricity consumers and/or producers offering and requesting advanced *energy* and *flexibility* products to/from the members (peers) of an EC.

P2P-FTP consists of 3 distributed blockchain applications (DAPPs) built on top of the Hyperledger Fabric – offering the functionalities for *identity management*, *pseudo-currency*, and *P2P trading.* The platform also offers an API for interfacing external systems, and it is to be deployed within a selected cloud environment with the help of a continuous integration and deployment (CI/CD) pipeline, which simplifies this process. The document has first presented simple and intuitive so-called user stories that explained the process of P2P trading from the end-user perspective. Later, these user-stories were used to derive requirements for the overall P2P-FTP and its DAPPs. To address these requirements, the specifications of the high-level P2P-FTP architecture, individual P2P-FTP DAPPs, external P2P-FTP API, P2P-FTP deployment environment, as well as CI/CD pipeline were given. The specifications presented in this document will guide subsequently practical P2P-FTP development efforts in WP5 (D5.2-D5.3).

11 List of tables

12 List of figures

13 List of references

[1] "DIRECTIVE (EU) 2019/944 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 5 June 2019 on common rules for the internal market for electricity and amending Directive 2012/27/EU," [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32019L0944.

[2] "Directive (EU) 2018/2001 of the European Parliament and of the Council of 11 December 2018 on the promotion of the use of energy from renewable sources," [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=uriserv:OJ.L_.2018.328.01.0082.01.ENG.

[3] "Open Source Blockchain Technologies," [Online]. Available: www.hyperledger.org.

[4] "Open Source P2P Money," [Online]. Available: bitcoin.org.

[5] "Community-run technology powering the cryptocurrency, ether (ETH) and thousands of decentralized applications.," [Online]. Available: https://ethereum.org/en/.

[6] E. A. e. al., "Hyperledger Fabric: A Distributed Operating System forPermissioned Blockchains".

[7] E. S. C. a. C. F. Androulaki, "Private and confidential transactions with hyperledger fabric," 2018.

[8] "Hyperledger Fabric Chaincode Lifecycle," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/chaincode_lifecycle.html.

[9] S. d. Quénetain, "Hyperledger Project: How to Make the Blockchain Flexible!!," [Online]. Available: https://www.blockchains-expert.com/en/hyperledger-guide/.

[10] S. S. S. F. S. B. R. H. W. P. D. Cooper, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," 01 05 2008. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5280. [Accessed 01 06 2021].

[11] V. B. Fabian Vogelsteller, "EIP-20: ERC-20 Token Standard," 19 11 2015. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20. [Accessed 31 05 2021].

[12] F. P. D. L. Julio Faura, "EIP-1996: Holdable Token," 10 04 2019. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1996. [Accessed 2021].

[13] a. a. io.builders, "Holdable Token," [Online]. Available: https://github.com/IoBuilders/holdable-token. [Accessed 01 06 2021].

[14] "Docker OS Virtualization," [Online]. Available: https://www.docker.com/.

[15] "Containerd: Container Runtime," [Online]. Available: https://containerd.io/.

[16] "Production-Grade Container Orchestration," [Online]. Available: https://kubernetes.io/.

[17] "Kubernetes: What is Kubernetes," [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[18] “Kubernetes Web UI Dashboard,” [Online]. Available: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.

[19] “Prometheus monitoring tool,” [Online]. Available: https://prometheus.io/.

[20] “Grafana monitoring visualizer,” [Online]. Available: https://grafana.com/.

[21] “Prometheus and Grafana integration with Kubernetes,” [Online]. Available: https://moizalimoomin.medium.com/integrate-prometheus-and-grafana-using-kubernetes-6b40e09e3ae7.

[22] “Elastic search engine,” [Online]. Available: https://www.elastic.co/.

[23] “Kibana data visualizer,” [Online]. Available: https://www.elastic.co/kibana/.

[24] “FluentD Open source data collector,” [Online]. Available: https://www.fluentd.org/.

[25] D. a. O. J. Ongaro, *In search of an understandable consensus algorithm (extended version),* Tech Report. May, 2014. http://ramcloud.stanford.edu/Raft.pdf, 2013.

[26] “Raft Protocol,” [Online]. Available: https://raft.github.io.

[27] “Etcd,” [Online]. Available: https://etcd.io/.

[28] “Hyperledger Explorer,” [Online]. Available: https://www.hyperledger.org/use/explorer.

[29] “Git - distributed Version Control System,” [Online]. Available: https://git-scm.com/.

[30] “github.com,” [Online]. Available: https://github.com/.

[31] “gitlab.com,” [Online]. Available: https://gitlab.com.

[32] “ICOM - Private GitLab,” [Online]. Available: https://colab-repo.intracom-telecom.com.

[33] “GitLab Runner,” [Online]. Available: https://docs.gitlab.com/runner/.

[34] Author, *Title,* City: Publisher, Year.

[35] Author, *Title,* Location, Year.

[36] *Grant Agreement No. 864537 – FEVER.*

[37] “Deploy EFK stack to Kubernetes,” [Online]. Available: https://medium.com/avmconsulting-blog/how-to-deploy-an-efk-stack-to-kubernetes-ebc1b539d063.

[38] “Kubernetes RBAC,” [Online]. Available: https://k21academy.com/docker-kubernetes/rbac-role-based-access-control/.

14 List of abbreviations

Abbreviation	Term
EC	Energy Community
P2P	Peer-to-peer
FTP	Flexibility trading platform
P2P-FTP	Peer to peer flexibility trading platform
DLT	Distributed ledger technology
HLF	Hyperledger Fabric – a blockchain platform, originally developed by IBM
FlexOffer	Flexible offer – a unified representation of a flexibility or energy product
DSO	Distribution system operator
VM	Virtual Machine
CI/CD	Continuous Integration / Continuous (Delivery, Deployment)
DAPP	Distributed Application
CLI	Command Line Interface
CFT	Crash Fault Tolerant
VCS	Version Control System