NGI LEDGER

Grant agreement 825268

An IT Asset Disposition platform that incentivises and certifies the circular economy of digital devices: Operative MVP – December 2019

Deliverable ID:	**D 1**
Deliverable Title: (generic)	Operative MVP – December 2019
Revision #:	**V1.0**
Dissemination Level:	**Public**
Responsible beneficiary:	NGI LEDGER
Contributing beneficiaries:	UPC, Pangea
Contractual date of delivery:	06.01.2020
Actual submission date:	04.01.2020

Start Date of the Project: 1 January 2019 Duration: 36 Months

Table of Contents

Approvals

	Name	Organization	Date	Visa
Project Management Team	David Franquesa	Pangea	2/1/2020	
PM team	Leandro Navarro	UPC	4/1/2020	

Document history

Revision	Date	Modification	Authors
0.1	31/12/2019	First complete draft	Stephan Fortelny, Jordi Nadeu, Sergio Mosquera, Emmanouil Dimogerontakis, Mireia Roura, Leandro Navarro, David Franquesa
0.2	2/1/2020	Update	Leandro Navarro, Sergio Mosquera, Stephan Fortelny
0.3	3/1/2020	Update and completion	Leandro Navarro, Stephan Fortelny, David Franquesa
1.0	4/1/2020	Consistency check and final reading (TBD)	Leandro Navarro

Operative MVP

Executive Summary

The implementation of a deposit-refund system (DRS) is the basis for our value proposition of facilitating the compliance with donor conditions during the entire life cycle of a computing device. Donors charge a deposit to refurbishers, who will in turn charge the deposit to final customers, and all these relevant changes are recorded in an irreversible ledger. If the donors' conditions are met, e.g. the device is recycled at the end of its life, and then the smart contracts will be triggered to pay back the deposit automatically and inexorably to all agents participating in the reverse supply chain.

The development of the operative MVP of an eReuse ledger, i.e. a permissioned Ethereum Proof-of-Authority blockchain, including a deposit mechanism has been carried out to implement the DRS as specified in the work plan. On a technical level, the deposit and conditions are represented as a smart contract attached to a device. The code of the MVP, in a public repository, implements this behaviour at TRL 4 (technology validated in lab), and the integration code with the DeviceHub web application to manage device inventories, and a video recording demonstrates how it can operate in practice, integrated with deviceHub at TRL 5 (technology validated in relevant environment). The operative MVP is the basis for further developments and pilot tests with early adopters and to define business models.

Work plan description

Operative MVP (6M - December 2019)

At the time of device transfer between an agent (donor) and an agent (refurbisher), the donor can establish a deposit. When the refurbisher accepts the transfer, they will be charged a deposit. Integrated with the backend and frontend of the Usody inventory system, these transactions are recorded in the eReuse ledger/blockchain, as part of the traceability record of the device. The implementation of the deposit system allows us to implement the take-back mechanisms, because if the device is returned and complies with the terms and conditions, then the deposit will be returned inexorably.

Future improvements - Market Proof of the MVP (M9 - March 2020)

Early adopters can register the proofs of disposal, data wipe, function, reuse and recycling in the blockchain:

1. **Proof of disposal**: After collecting the devices from the donor, the refurbisher generates with the software an inventory of these devices. This creates a *secure disposal delivery note*. The donor confirms the delivery note and sets the required deposit. At this point, the deposit is billed to the refurbisher and the transfer is recorded in the blockchain.

2. **Proof of data wipe**: The donor or the refurbisher uses the software to generate a certificate of data wipe. This information is recorded in the blockchain and donors have access to it.
3. **Proof of function**: The proof of function can be performed any time with a software that runs on the device and creates a rating of its performance. The refurbisher can share this information with potential buyers.
4. **Proof of reuse**: The final consumer accepts the transfer of the device created by the refurbisher, and at this point the deposit is billed to the final consumer. This transfer is recorded in the blockchain.
5. **Proof of recycling**: After recycling the device the final consumer provides information of the collection point (name of the collection point, time of delivery, contact person and, if possible, a delivery ticket). This information is recorded in the blockchain. At this point the deposit will also be returned to the final consumer.

1. Implemented user story

The operative MVP allows a full transfer of device ownership between two agents, including the transfer of a deposit using Ethereum tokens. Through the newly implemented functionality, integrating the blockchain with the existing inventory system, users:

- share groups of devices (called lots) with each other,
- accept the transfer of devices,
- enter a deposit that is withdrawn automatically from the recipient's account on transfer completion.

1.1 Screenshots

Through the following screenshots, a complete transfer of devices is shown:

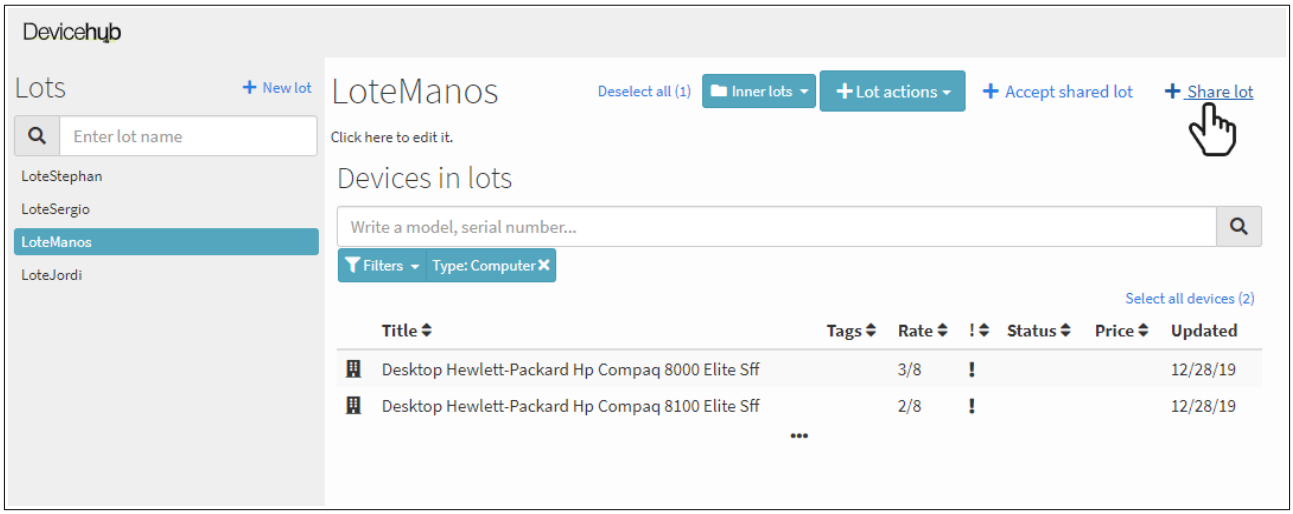
1. View registered device that will be shared
2. Sharing several devices (user 1, current owner)
3. View registered device after it has been shared
4. Accept transfer of device and enter deposit (user 2, recipient)
5. View registered device after it has been transferred

View registered device

Note that *owner_address* is set to the blockchain address of the device's current owner (*0xC79F7fE80B5676fe38D8187b79d55F7A61e702b2*)

Sharing devices 1

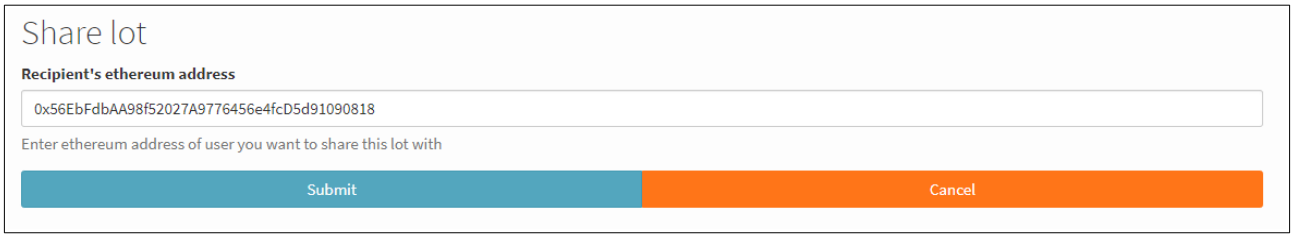
An owner of devices selects a lot, i.e. a group of devices, and shares it with another user.



Sharing devices 2

The owner specifies the recipient‘s blockchain address.

Note: In the future, an integrated search or address look-up should facilitate entering this information.



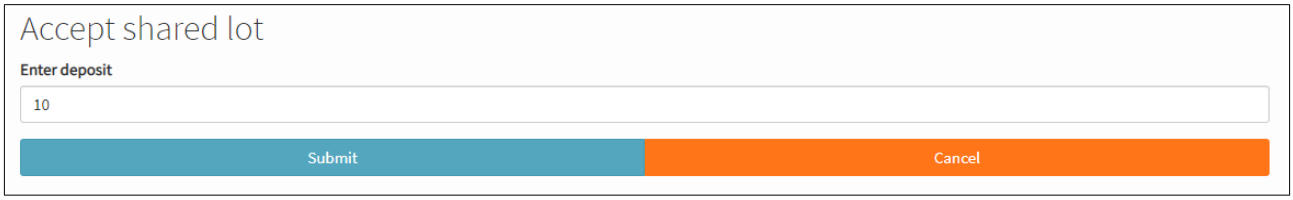
Public link of device after sharing its encompassing lot

The properties *receiver_address*, indicating that the device has been shared with another user, and the *deliverynote_address*, indicating that the device is linked to an encompassing lot, are set.

Acceptance of a transfer

The receiving user accepts the shared lot, including all included devices, in exchange of a stipulated deposit. The (per unit) amount of the deposit, multiplied by the number of transferred devices, is withdrawn from the receiving user.

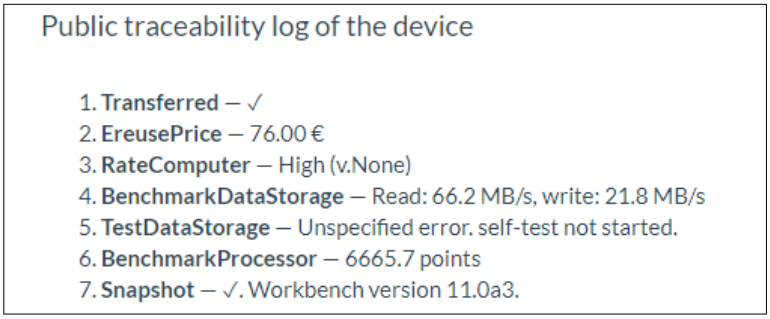
Note: In a future enhancement, the deposit should be specified in advance by the first user and confirmed by the second.



Update of public link including traceability log

When the transfer is completed, through the acceptance of the receiving user, the ownership is changed, and the change recorded in the blockchain, as well as on the device inventory server.

The device's public traceability log, is also updated with the *Transferred* event.



1.2 Screencast

The screencast, available at https://ledger.dyne.org/s/CNZHZX7bmNPwemj[1], demonstrates how an employee of a public administration (user@dhub.com) can offer a device (a desktop computer) to a refurbisher (user2@dhub.com), and perform the transfer of ownership, leaving a deposit (10 currency units) as an incentive and guarantee of circularity (traceability of the device across lifespan and final recycling).

The system offers each organization their own private interface to manage their devices (including the transfer of them).

There is also a public open interface to see public reports of a device, that shows information made public about that device for transparency requirements set by the participants. It includes the characteristics of the device, including ownership, associated deposit and track of changes, linked to the ledger (blockchain) infrastructure. This public report reflects changes on the ownership of the device, its status and associated deposit.

The system shows a web interface to the DeviceHub web application, integrated with a PoA Ethereum permissioned blockchain where key events are recorded and run Solidity smart contracts to manage the contractual obligations associated to devices, including deposits.

1 As a download link: https://ledger.dyne.org/s/CNZHZX7bmNPwemj/download and with subtitles optional https://ledger.dyne.org/s/QTJZop7aQGCws7a

2. Technical documentation

We describe in Section 2.1 the overall or general architectural principles of the system of a blockchain-based backend with a web application (DeviceHub) that a) stores private data about devices linked to owners, organizations and history, b) acts as a front-end to the blockchain backend, maintaining privacy and confidentiality preserving links between private detailed data and public summarized data through summaries, and c) offers a user (web) interface to human users to manage this private data, and to software tools (e.g. Workbench) that run on physical devices to extract and report data and status of hardware devices. For that, Section 2.2 describes our approach to data collection and processing. Section 2.3 describes the specific software architecture of the MVP implementation, the interactions across elements, and links to the source code repositories.

2.1 General architecture

A backend based on blockchain technology is integrated with the existing system, a monolithic web server including API and database as well a corresponding web client. Important transaction data, such as device ID and deposit amount, is recorded on the blockchain data, so it can work as an append only, irreversible log, and later be verified through different clients, over different replicas, and by third parties. The web server stores references (Ethereum addresses) in the blockchain. Additionally, some parts used for the device transfer, e.g. amount of deposit, are stored both on the web server and recorded in the blockchain replicas, since this information is used for visualization to the user as well as for verification purposes.

The main purpose to use Blockchain technology is to provide immutability (append-only or irreversibility), security, transparency in the data, and inexorability of the actions (conditional transfer of deposits, unstoppable, even in the future). This is crucial for eReuse, to have a way to promote the traceability of the different actions that are carried out among any actors of the system, that may be involved in the usage of one or more devices in a circular economy.

Therefore, the web client is a user interface that manages private information (detailed for owners and managers with access permissions) and public information (for anyone willing to access the public access limited information, for traceability and accountability purposes). The web client is also an interface to the blockchain backend that summarizes and records data of the replicated blockchain data structure, and runs a set of smart contracts.

A set of user requests made through the web client are sent directly to the blockchain data store. Hence, they can not be modified by the web server, and blockchain features such as immutability and security are guaranteed. Additionally, another client (e.g. from OBADA) may be used independently from the DeviceHub web server to verify blockchain data.

2.2 General approach to data collection and processing

Our project focuses on a distributed ledger (blockchain), that could be implemented either on a public (permissionless) or private (permissioned) one, which results in more or less visibility of the data stored in that blockchain (to everyone in the first case, to the members of a consortium in the second). Our preference is for a private (consortium) ledger.

In the ledger we store attestations about the data (hashes), not the data itself. Those that have the information can prove it from attestations (data) stored in the ledger, not the other way around.

These attestations of the data rely on unidirectional functions that represent the two involved components: Devices and Actors (Humans or Entities)

Device/component IDs are linked to external (protected) data about devices/components using hashes in URL/URNs (or in some cases keyed hash functions, with different properties).

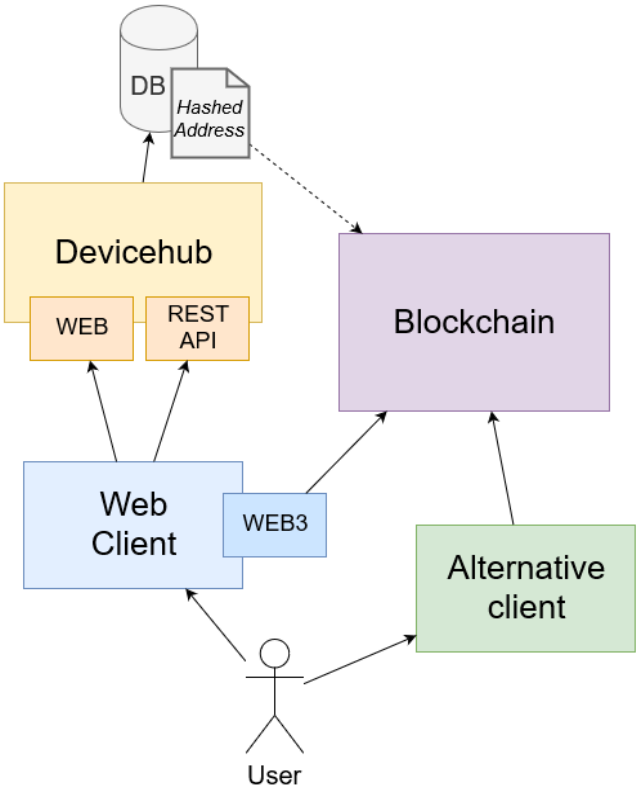
Regarding (human) user identifiers (participants), we use Ethereum-like addresses. Although these addresses do not contain personal information, they are leak sensitive, in the sense that if the identity of its owner is disclosed, we could identify its participation in all transactions in the ledger, including the number of devices it has, that allows to guess its wealth or position in the market. Given that repeatability of participant IDs, we use an external entity, a registry in OBADA terms, that generates unique identifiers for each transaction, therefore obfuscating the participants of the transaction, without damaging the auditability property offered by the blockchain. The current MVP implementation does not include the registry function to protect from repeatability of participant IDs and enhance data confinement, and therefore a potential covert channel[2].

In simple terms, the ledger does not store any personal data, just proofs of it, and thus only meaningful for those that already have it.

2.3 Specific software architecture implemented

The architecture of the main building blocks of the MVP software in the next figure, that shows how interactions can pass through a web interface to a) the private device data repository offered by the DeviceHub server (via HTML Web for humans, or the REST API for tools such as Workbench), and b) to the blockchain backend (via the web3 library). Alternative clients that can directly interact (read or write) with any replica of the eReuse blockchain. The following diagram also shows how the private device data and events stored in DeviceHub are kept in sync with what is published in the blockchain through privacy/confidentiality preserving summaries (hashes):

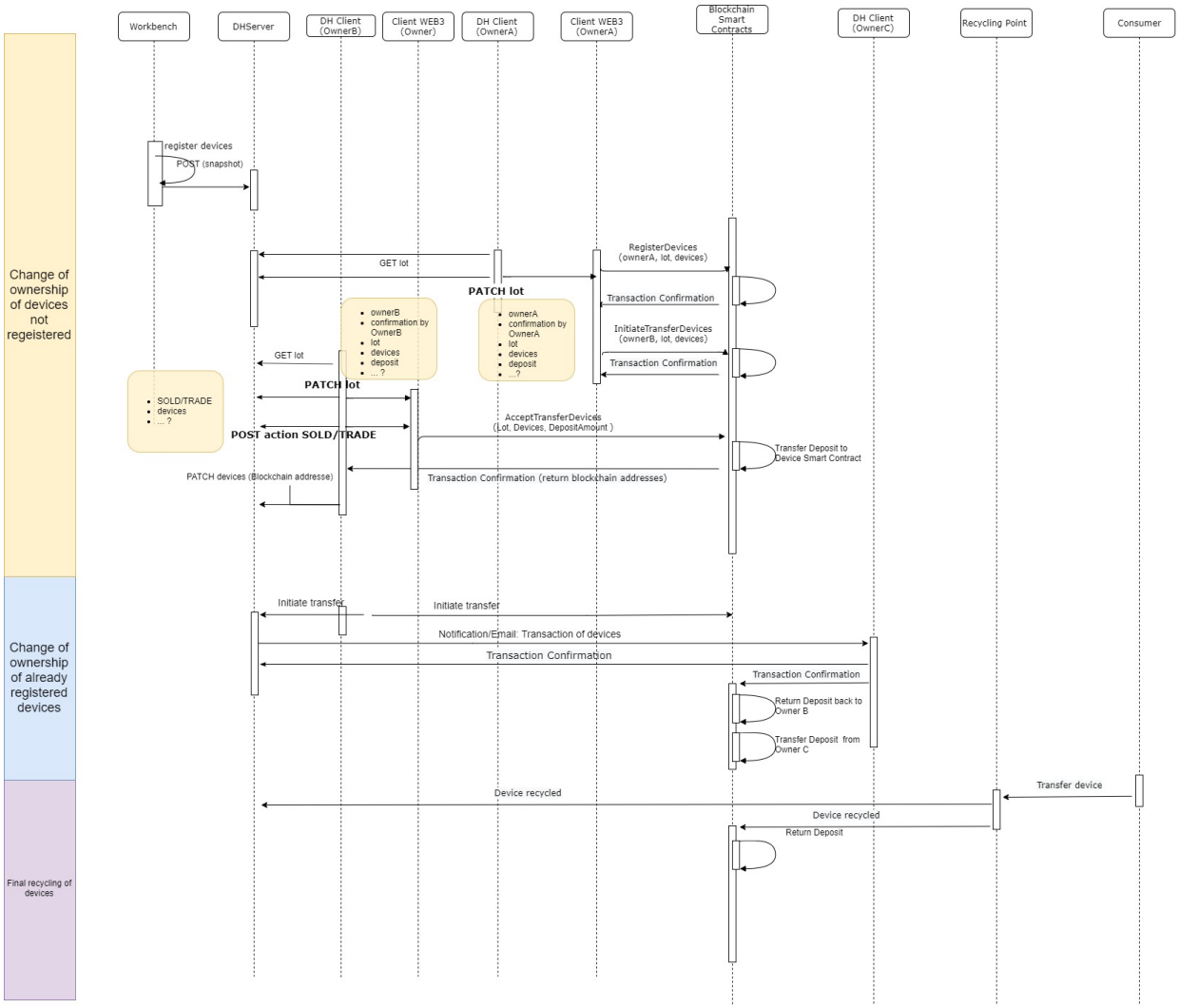
2 Also known as a Lampson channel, https://en.wikipedia.org/wiki/Covert_channel



Please find the source code repositories in these URL:

- Web client: https://github.com/eReuse/DeviceHubClient/tree/deliverynote
- Blockchain specifics: https://github.com/eReuse/DeviceHubClient/tree/deliverynote/app/truffle
- Web server: https://github.com/eReuse/devicehub-teal/tree/deliverynote

The following sequence diagram shows the main behaviours, with the flow of control and data among the elements in the system, and how elements and actors communicate with each other with sequences of messages:



The following sections provide details about the main elements of the system and their design.

2.4 Devicehub client and server

Devicehub is an IT Asset Management System focused on reusing devices, created under the project eReuse.org.

Devicehub was initially set up as a distributed system, with each instance being an inventory and having its own separate database. Through this project, the architecture was changed to a centralized server to facilitate sharing of devices between users.

Devicehub is a monolithic web server, with web hosting and API sharing the same code base and hosted on the same instance. The REST API of DeviceHub is built with Teal and Flask using PostgreSQL. DevicehubClient is the front–end that consumes the API.

Data structure

Devices are the main entity in a Devicehub inventory. Lots are folders containing devices and other lots. One lot can be contained in several lots.

Actions are events performed to devices, changing their state. Actions can have attributes defining where it happened, who performed them, when, etc. Actions are stored in a log for each device. Devicehub actions inherit from schema actions.

More information available at http://devicehub.ereuse.org/

2.5 Blockchain-related elements

This section covers the design and development of the blockchain environment and is divided into different sections to explain: the considerations about the scope of blockchains (public to any number of participants or limited to an allowed group (permissioned), the technology choices to implement it, the main smart contracts implemented and the interaction with the rest of the system, and finally the operations that are carried out within the blockchain in an end-to-end interaction.

2.5.1 Blockchain considerations and choices

Blockchain technology offers solutions that seems apt to make the peer-to-peer nature of access networks trusted and economically sustainable. Blockchain is an immutable and distributed data storage without the provision of retrospective mutation in data records. However, most blockchain networks are open and public (permissionless) that encourage the users to protect anonymity. This implies that anyone, without revealing their true identity, can be part of such a network and make transactions with another similarly pseudonym peer of the network.

Permissioned blockchains

Because of such registration process there is also the need for an efficient identity mechanism on top of blockchain's immutable record keeping. Permissioned blockchains are part of such solutions,

mostly envisioned for business networks where there is often a stringent requirement of knowing your customer in addition to keeping the intra- and inter-business transactions confidential.

The requirement of both users' identity and trusted record keeping is of paramount importance and that is why we decided to use a private permissioned blockchains. Hyperledger Fabric fulfils by default these properties. In contrast, while the Ethereum software is not primarily destined to serve these purposes, it can also be used to operate a private permissioned blockchain. Some protocols, like Ethereum, offer inexpensive consensus algorithms for known and bounded membership, like the Proof-of-Authority (POA) protocol, that are ideal for a private permissioned instances, as envisaged in our scenario.

Ethereum Protocol

In the Ethereum ecosystem[3], there are two main types of entities namely: i) an externally owned account (EOA) with an address and a ii) smart contract written in a contract-specific programming language, such as Solidity, that is compiled into byte code which gets executed by an EVM. In addition to an EOA, a smart contract is also assigned an address when it is deployed on the blockchain, however, it is used in a nuanced manner when compared to the address usage of an EOA. Anyone in possession of an EOA's address credentials can make a value-transfer transaction with another EOA by specifying its blockchain address. In such transfers the overall system state of the EVM remains unchanged. However, in contrast, it is also possible for an EOA to make a transaction with a smart contract. In these types of transactions a specific function of a smart contract is invoked that usually triggers a state change in the overall EVM. It is also possible that one smart contract invokes a function of another smart contract possibly executing another associated EVM. It should be noted here that in Ethereum, each time a piece of code is invoked for execution (such as a smart contract's function) all the nodes of the network execute the same piece of code ensuring the correct execution of a program's logic. The state change, in turn, is then recorded in a decentralized manner in the form of mined (more on mining later), which are mutually agreed-upon, blocks ensuring immutability of such records. This way Ethereum enables a trusted and decentralized environment to automate a consortium-based application with trusted value-transfer transactions among the (potentially mutually non-trusting) peers of such a consortium.

One of the other consensus engines currently in use in Ethereum's universe is called Clique. Clique engine makes use of a consensus protocol called Proof-of-Authority (PoA). In contrast with PoW, PoA is computationally less expensive and eases the process of scaling a network. PoA-based consensus engines help to establish a private and permissioned version of a blockchain. In PoA, in contrast with PoW, the nodes who can have a say in appending new blocks to a blockchain are carefully chosen with known identities are referred to as sealers. In turn, the process of appending a new block to a blockchain running a PoA-based consensus engine is called sealing. Such nodes are also sometimes referred to as authorities. Specifically, sealing implies that if a block contains

3 Kabbinale, AR, Dimogerontakis, E, Selimi, M, et al. Blockchain for economically sustainable wireless mesh networks. Concurrency Computat Pract Exper. 2019;e5349. https://doi.org/10.1002/cpe.5349

digital signatures of majority of authorities then it is considered as a valid block. However, PoA has proven to be less secure as compared to PoW and that is the reason that it is predominantly being used by test networks and private chain setups for experimental purposes.

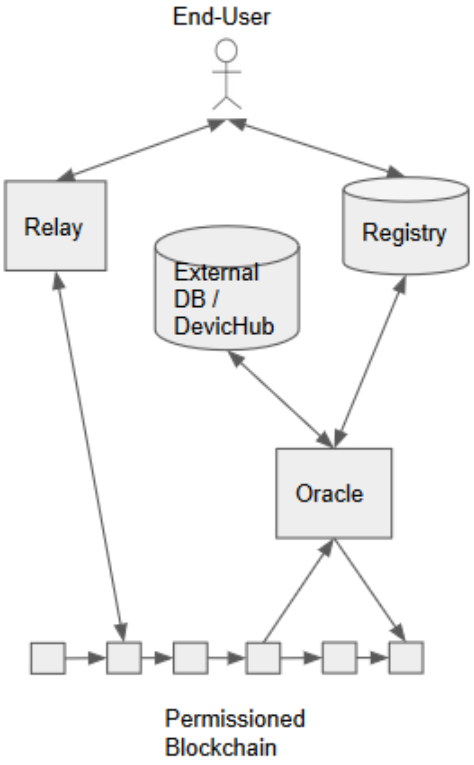
In summary, we have externally owned accounts and smart contracts and an Ethereum-based lightweight consensus protocol (permissioned and low overhead, PoA, as explained above) that constitute a private permissioned blockchain with several replicas, running smart contracts implemented in the Solidity language. To ensure decentralization, and global properties such as integrity, availability, transparency and accountability, this blockchain can be run by multiple organisations member of a consortium interested in ensuring the circularity of digital devices. A possible alternative to this approach could be Hyperledger Fabric.

2.5.2 Blockchain system architecture

Our design relies on a local blockchain infrastructure that operates over an Ethereum-based API running on a set of local Ethereum servers.

Main components

Each device is represented as a **Smart Contract** in the Blockchain. The **Registry** is responsible for anonymity and authorization. The **Relay** allows End-Users to participate without being connected directly to the Blockchain. An **External DB** stores sensitive device information



User transaction flow

End-Users obtain transaction tokens from the Registry.

End-users send signed transactions to the relay.

The Relay forwards transactions to a permissioned blockchain.

The blockchain verifies transaction validity with the registry.

If necessary the External DB is consulted or updated.

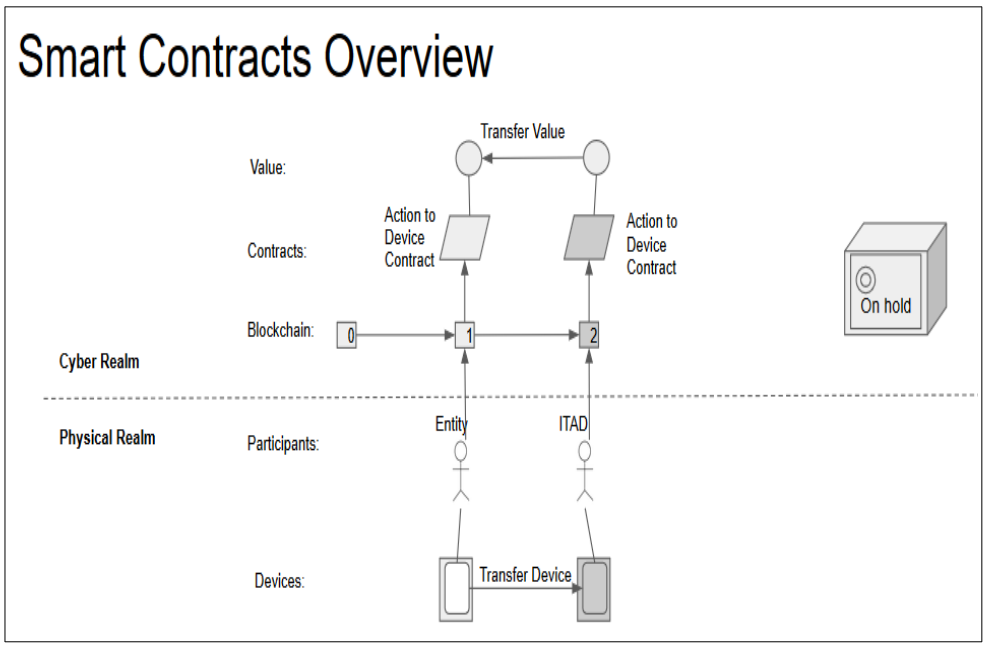
The Blockchain executes and records the transaction.

We have set up an Ethereum node, which is loaded with the set of smart contracts used to model the system. These smart contracts hold the definition of the operations that are needed to mimic the system behaviour and to provide the necessary features (data immutability, security, transparency).

2.5.3 An overview of the Smart contracts involved

The main building block of our architecture is a smart contract implementation of a *conditionally transferable device deposit*. A set of smart contracts represent devices and an associated "wallet" containing the following information:

- *some roles or identities*
- *some conditions: state of device, existence or absence of the chain, in which part/frame of the chain is a value*



In terms of deployment, we have evaluated two different options:

1. A centralized ERC721 contract, responsible for evaluating the conditions and applying the necessary actions for all the devices and their conditionally transferable deposits *(not very scalable for big number of devices)*
2. The ERC721 contract as an index for a single smart contract that is responsible for evaluating the conditions and applying the necessary actions of each concrete device.

We use two tokens, an irreversible log (blockchain), and smart contracts to implement the inexorable execution of contractual rules:

1. One is a fungible token (cash, an amount of currency), an ERC20 compatible interface in Ethereum terms. A wallet is an address/ID pointing to a data structure that keeps track of the balance of each account.

2. Another is a non-fungible token or NFT (ownership, ID of linked device), ERC721 compatible interface in Ethereum terms. An NFT ID points to a data structure for that device including a link to the URL (in DeviceHub), amount of deposit, owner's address, etc.

3. The (permissioned) blockchain as the crypto-chained log file with entries for each operation.

2.5.4 Dependencies

In order to implement the system in the best possible manner, we build on Open Source libraries and applications that ease significantly this task:

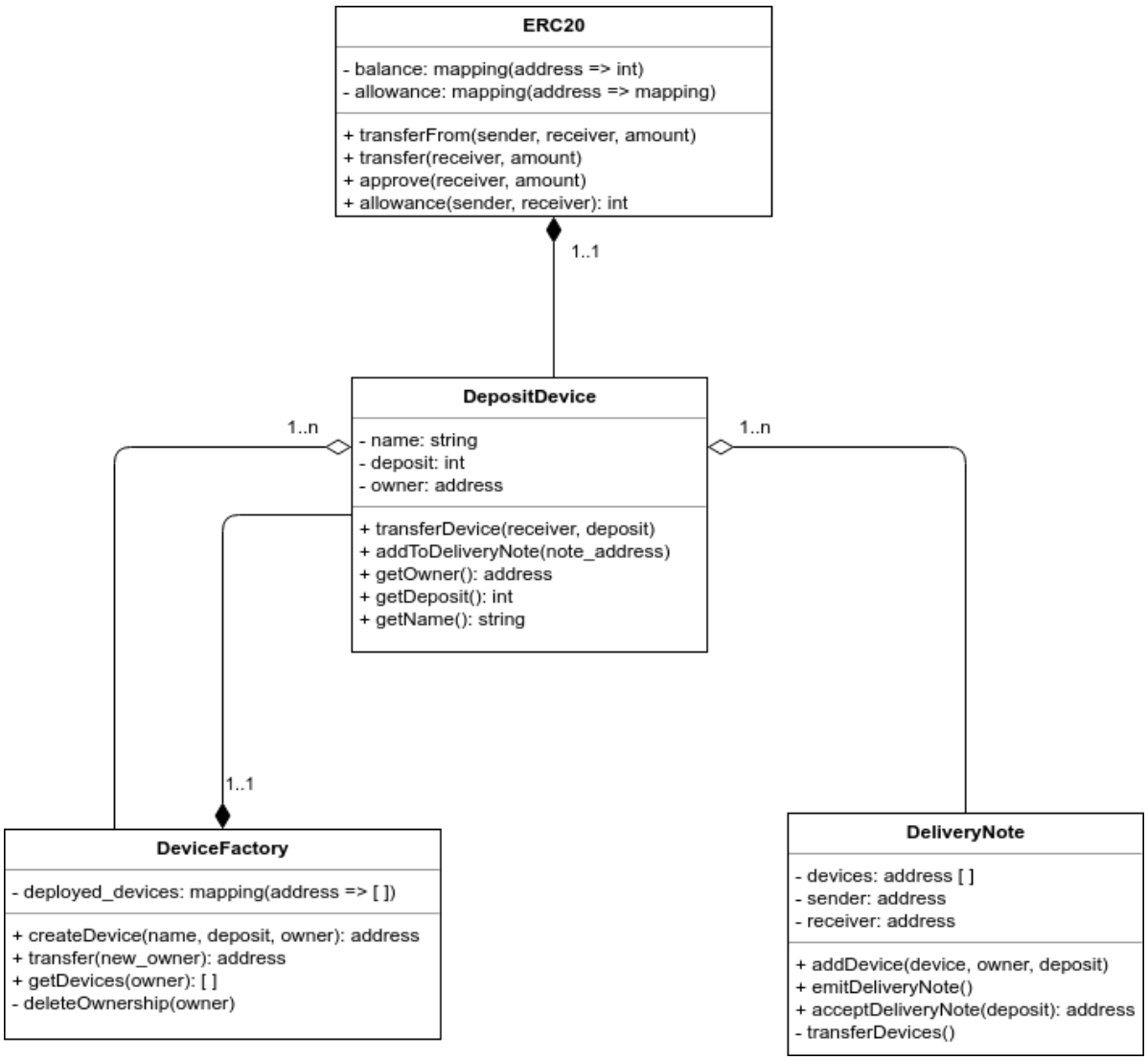
- **Solidity**. Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs that are executed inside a peer-to-peer network where nobody has special authority over the execution. These contracts are executed against the Ethereum Virtual Machine (EVM), that is not only sandboxed but actually completely isolated, which means that code running inside the EVM has no access to network, filesystem or other processes.
- **Solidity Compiler**. In order to execute some smart contract operations, the EVM needs not the definition we implement of the contract, but the binaries that are obtained as an outcome of the compilation process. That is the reason that we need to compile the contracts whenever we introduce a new feature or bug fix.
- **OpenZeppelin**. Is a set of libraries that provide the capability of writing, deploying and operating decentralized applications in safe way. All these libraries are Open Source and they have been very useful for the implementation of the system, specially on the ownership management part.
- **Truffle Suite**. It encloses a set of development tools that are a must for blockchain developers. For the system we have used two of these tools:
    - **Truffle**. The most popular development framework for Ethereum, provides built-in smart contract compilation and deployment, network management for deploying to any number of public and private networks, automated smart contract testing, or even an interactive console for direct contract communication.
    - **Ganache**. Is a personal blockchain for Ethereum development that developers can use to deploy contracts, develop applications, and run tests. Ganache comes with a set of predefined Ethereum accounts (which is the unique identifier for the users within the blockchain) so that you can start testing your application from the very beginning.
- **Web3.js**. As it was mentioned on the *Solidity* description, the EVM is completely isolated. This is an issue when we want it to provide communication with external systems. Web3.js takes control of the communication between the system and our Ethereum blockchain using a protocol called JSON RPC, which stands for *Remote Procedure Call* protocol. Web3.js makes possible to make requests to a specific Ethereum blockchain in order to read and write data to that network.

2.5.5 Smart contracts implementation

This section is intended to describe the blockchain environment with more detail. That is, explain the smart contracts implementation and the interactions among them, as well as the integration with *DeviceHubClient* and how the communication between both systems is performed.

Contracts

In OOP (Object Oriented Programming), the abstraction of real life systems is done by splitting the responsibilities of the system into several classes. In Solidity, these classes are substituted by the Smart Contracts, which are capable of executing operations, communicating with other contracts or even creating an inheritance structure with them. To model the behaviour of the system we have created four main smart contracts: *DeviceFactory*, *DepositDevice*, *DeliveryNote* and *ERC20*, which will be explained with more detail in the following sections.



Device Factory

First of all, owners of devices should be allowed to register the devices into the system, so creating and storing them into the blockchain. Following the definition of the problem, we find out that the Factory design pattern fits perfectly to solve it. So that is the main idea behind the *DeviceFactory* contract, a contract whose main task is to create and keep track of the devices belonging to each owner.

This contract stores in form of a *mapping* (key-value storage) the different devices (value) that have been registered from some user's Ethereum address (key). This contract will be also used to backup and centralize the ownership information from the user point of view, in such a way that a user would be able to check which are the devices that they own at some point. In order to do this, the collection of devices for each user needs to be updated every time an operation regarding the ownership of some device is executed.

Deposit Device

According to the above image, there is a bidirectional interaction between this contract and the DeviceFactory. As explained in the previous section, DeviceFactory stores a set of devices identified by the address of their owner. The attributes and operations that can be performed over these devices are modelled using the *DepositDevice* smart contract. The main attributes stored on each device contract are the string identifier for the device (usually the model of the device), the Ethereum address of its owner, and the proportional part of the deposit done for every device in a lot. It is important to remark that the deposit value inside the blockchain represents a number of ERC20 tokens (for sake of simplicity a unit of money is mapped to one token).

DepositDevice contract will hold all the logic related to the ownership transference from a single device, that is every device controls its own ownership chain. To manage this ownership we take advantage of the Ownable contract provided by OpenZeppelin. The ownership transference consists of four main steps: add the device to the corresponding delivery note; return the deposit held by the device to the former owner; update the owner associated to the device by the new one, as well as the deposit chosen by the new owner; update the ownership information contained in the DeviceFactory contract. All this operations are performed from within the Device contract.

Delivery Note

The devices are grouped into lots that later are transferred to a new owner which states an amount of money to be deposited. To model the behaviour of these lots we have decided to create the *DeliveryNote* contract such that it will hold the set of devices that will be transferred from its current owner (sender) to a new owner (receiver). In order to complete the task of ownership transference inside the blockchain (we remark this because the operations within the blockchain do not following exactly the same steps as in the server side, but the outcome is the same), four main tasks have to be fulfilled: the original owner has to create the delivery note; each device has to be added to the delivery note; once every device is in the delivery note, this note is sent to the new owner; the new owner has to accept the delivery note by setting a deposit price.

The life-cycle of a delivery note is short, and finishes when the ownership transference is completed. The steps explained before match with the ones explained in the Deposit Device section, for instance, once the receiver of the devices accepts the note, the deposit is returned to the former owner and the ownership is updated both in the device and the device factory contracts.

EIP20

Managing fungible actives that are treated as money is a hard and subtle task. This is a common situation in blockchain and luckily there are some standard interfaces to handle it. The most common abstraction is to think of these values as tokens, being one of the most common interfaces the ERC-20 standard and the most reliable and stable version of this implementation is maintained by OpenZeppelin. This implementation makes possible not only the transference of tokens between Ethereum accounts, but also allow tokens to be approved so they can be spent by another on-chain third party.

This is an important improvement because the task of transferring tokens from one account to another relies on the devices itself, so they need to have some allowance to make it possible.

2.5.6 Integration with the client

Once we have the base implementation of the smart contracts we need a way to interact with the client, that will be the module that will communicate directly with the blockchain. At this point we will be using some of the tools explained in Dependencies section, mainly **Truffle** and **Ganache** for the contracts compilation and deployment, and **Web3.js**, to provide a communication between the deployed contracts and the frontend.

Contracts deployment

For the smart contracts to be interactive from the client, we need to deploy them to some Ethereum node. The truffle suite offers the Ganache client that makes possible to set up an Ethereum node in matter of seconds. Along with Ganache, the Truffle library provides tools for the compilation and deployment of the contracts to some blockchain node (configuring beforehand the target of our deployment).

An important remark is the difference between compilation and deployment, as not every smart contract is deployed initially. Every smart contract is compiled as it needs to be interpreted by the EVM, but only some of them are deployed at the beginning as specified in the migrations. A common refactor that is done to the deployment process is saying migrations instead. The idea behind the migrations is to have a set of smart contracts working from the beginning as they do not need to be created dynamically. An easy example would be the difference between the DeviceFactory and the DepositDevice, as the former one is included in the migrations while the late one is created dynamically during the execution of the system. The point is that we need a single DeviceFactory throughout the whole execution, that is the point of deploying it at the very beginning, while for the DepositDevice we do not need it until we want to register a device, so it makes no sense to have it in the blockchain before that creation takes place.

Communication with the blockchain

The outcome of the contracts compilation is a set of binaries that represent the smart contracts in such a way that the EVM can interpret it. While the deployment process provides a unique Ethereum address to each instance of the deployed contracts. Using both results, Web3.js is able

to create an instance of the deployed smart contract to interact with it and to make persistent changes on the Ethereum node. Web3 offers several distributions of the library regarding the programming language we will be using, but the most mature one is the .js version. As the client has been developed using React.js, we are using the .js version. From this point, and just using Web3.js and the binaries obtained during the compilation process we are able both to obtain instances of already deployed contracts and to deploy new contracts.

2.5.7 Operations

There are a set of functional steps to be followed in order to achieve the final output, which is the transference of the (registered) devices from a given owner (OwnerA) to other one (OwnerB), whilst this last one sets a deposit on the devices to ensure that they are either recycled or transferred to subsequent users. The steps are the following:

Register devices

When we receive a set of devices that will be included in the Delivery Note, this set of devices needs to be inserted in the blockchain. This have to follow an ordered pipeline to ensure the validity of the data (we are using the account of OwnerA):

- We get instances of the smart contracts that we are using in this process using *Web3.js*. In this case we need the DeviceFactory to create the devices, and the DepositDevice contract, which will represent the information of each of the devices to be transferred. The creation of the devices is delegated into the DeviceFactory client, but to read the information about any of these devices we need to use the DepositDevice contract.
- Once we have the needed instances, we will create each of the devices to be transferred. This operation needs to be started from the OwnerA account such that the ownership of these devices will be assigned to them initially.

Create/send delivery note

The next step would consist in creating the delivery note that will be sent from OwnerA to OwnerB. We are still using the account of OwnerA at this point.

- In this case we do not need to get instances of deployed contracts, but to create a new one, a new DeliveryNote. For this reason we need to use the binaries of the compiled contract to create a new instance of this contract in the blockchain. We just need to specify the receiver (OwnerB) address as the sender one is obtained immediately, as this operation can only be performed from the owner of the devices.
- After the creation we need to add every device that will be transferred to the note and, again, this operation can only be performed from the account of the delivery note sender.
- Last step would be to emit (send) the delivery note to its target user. This operation allows the receiver to be the only one that can accept the device transfer.

Accept delivery note and transfer devices

The last part, although it just means a single operation for the user has a lot of logic behind it. Take into account that we need to transfer both devices and tokens between users. At this point we are using the OwnerB account:

- We receive as input for this operation the DeliveryNote address and the deposit done by OwnerB to accept the devices transfer.
- First step consists in getting an instance of the existing delivery note from its Ethereum address. This will let us interact with this delivery note.
- Then, as this deposit will be set, we need to *approve* this tokens' transaction from the OwnerB to the devices in the delivery note. This is done by the ERC20 contract.
- Next step consists in returning the old deposit done by OwnerA from previous operations over the devices (again done by the ERC20 contract).
- Finally each device is transferred from the former owner to the new one, stating in each of them the proportional part of the new deposit done by OwnerB, and updating the information in the DeviceFactory contract.

3. Conclusions

We have designed an operative MVP of an eReuse ledger, including a traceability log of computer devices across their lifespan and different status and users, and a deposit mechanism, that implements a deposit-refund system (DRS), that is the basis for our value proposition of facilitating the compliance with donor conditions during the entire life cycle of a computing device. Donors charge a deposit to refurbishers, who will in turn charge the deposit to customers, and all these relevant changes are recorded in an irreversible ledger, distributed across several eReuse stakeholders (permissioned). If the donors' conditions are met, e.g. the device is recycled at the end of its life, and then the smart contracts will be triggered to pay back the deposit automatically and inexorably to all agents participating in the reverse supply chain.

The document describes different aspects of the design, implementation and validation of the MVP. The code is in public repositories and the implementation validated in laboratory conditions (TRL 4), and integrated with the DeviceHub web application in the relevant environment of management of device inventories, with a video recording that demonstrates how it can operate in practice, at TRL 5 (technology validated in relevant environment). The operative MVP is the basis for further developments and pilot tests with early adopters and to refine the business model.

Future improvements are planned to enable and get closer to a future market proof of the MVP, with a demonstration (TRL 6-7) in a relevant or operational environment, where early adopters can register the proofs of disposal, data wipe, function, reuse and recycling in the blockchain, and create human-centric solutions to manage electronic and computing devices that are economically,

environmentally, and socially sustainable and scalable, under the principles of the circular economy.

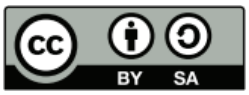
– == o O o == –

An IT Asset Disposition platform that incentivises and certifies the circular economy of digital devices:

Operative MVP – December 2019

NGI LEDGER

UPC, Pangea



Barcelona, December 2019