interlace_

Interacting Decentralized Transactional and Ledger Architecture for Mutual Credit

WP3

Iterative Demonstrator Implementation

Deliverable D3.2

Final Demonstrator Implementation

Horizon 2020

Project funded by the European Commission
Information and Communication Technologies

FET OPEN Launchpad Project
Grant no. 754494

Contract Number: 754494
Project Acronym: INTERLACE

Deliverable No: D3.2
Due Date: 31/10/2018
Delivery Date: 31/12/2018

Author: Eduard Hirsch (SUAS), Paolo Dini (UH), and Maria Luisa Mulas (SARDEX)
Partners contributed: Giuseppe Littera, Luca Carboni (SARDEX)
Made available to: Public

Versioning		
Version	**Date**	**Name, organization**
1	08/10/2018	Paolo Dini (UH)
2	31/10/2018	Eduard Hirsch (SUAS)
3	31/12/2018	Eduard Hirsch (SUAS), Paolo Dini (UH)

Internal Reviewer:	Paolo Dini (UH)

Abstract

This report describes the Hyperledger implementation of a part of the specification of the INTERLACE transactional platform as detailed in deliverable D3.1: Requirements and Architecture Definition, as well as mechanisms used to ensure a stable and shared runtime environment and to guarantee testability and easy execution of the ASIM model of the business logic. The implementation is based on a refinement of the requirements that is detailed in D3.1, along with an updated formal specification, relative to the original ASIM definitions of D2.1, that can be found in D2.3: Final Architecture. Finally, a presentation of the runtime environment is given and discussed in the context of the future connection to the blockchain-based backend.

Table of Contents

Chapter 1

Introduction

Eduard Hirsch

1.1 Objectives and Motivation

During the INTERLACE project, requirements for the transactional platform of an interest-free mutual credit system were collected and documented in deliverables D2.1[3] and D3.1[8]. The requirements were formulated precisely using “Abstracts State Interaction Machines" (ASIMs). These ASIM specifications were used to create a working ICEF implementation,[1] which also has been described in D3.1.

The focus of the present report is on the creation of a working prototype that provides basic payment capabilities, thereby enabling a scalable implementation of a distributed ledger technology (DLT).

1.2 Scope and Organization

This report provides insights into the prototype created. It discusses the technologies used and provides feedback on how they have been applied in order to achieve a working implementation capable of accepting basic transfers.

Further, information is given on how the system can be installed and used for paying. The system can therefore be used to prove and test various use cases in order to understand the chosen DLT approach.

For production systems, which might be built upon the INTERLACE prototype, important features are discussed that need to be handled during the realisation of such a payment system. Thus the transition from a client-server to a p2p-based model together with possible challenges are described and example approaches explained.

Finally, an outlook is given which elaborates possible scenarios that may be applied in a production context.

[1] http://biomicsproject.eu/news/135-icef.html
https://github.com/InterlaceProject/icef

Chapter 2

Design Discussion

Eduard Hirsch and Paolo Dini

2.1 ASIM and blockchain

Creating a scalable distributed application is a big challenge. Even more so when monolithic legacy systems need to be addressed as in the case of the Sardex payment system, partly realised in Sardinia. For INTERLACE this meant dealing with a system which is currently stable and working reliably. One of the drawbacks of this system, however, is that it does not scale well, which has become increasingly evident in recent times. This problem has become increasingly evident as more circuits from other parts of Italy have been added to the database.

This report describes a prototypical solution of how distribution and scalability are handled using technologies that allow a mutual credit system to grow and evolve. More specifically, for INTERLACE a well-tested architectural, specification, and modelling process was introduced, the Abstract States (Interaction) Machines paradigm [2], which uses a distributed computational model that allows for iterative refinement of different concurrent and communicating system components in a very specific and detailed way.

As described in [3], an ASIM definition was developed which acts as a ground model for the INTERLACE prototype. This paper-based definition was then transformed into executable code realised with the ICEF,[2] which is based on ASM language primitives [2]. The implementation, which is the main focus of this report, is the last step before testing, which needs to be done on various levels (component and field testing).

In the following subsections the various topics and challenges encountered during the planning and the definition of the new blockchain-based system are addressed.

2.1.1 From Servers to Agents and Peers

INTERLACE encourages not just a change in technology but also an architectural culture change. Currently many systems in industry are based on monolithic approaches which are stable and based on commonly known and widely adopted implementation strategies. Often not even based on multiple tiers, such classic strategies suit the needs of small and middle-sized projects but come at high cost for very large application services and their providers. When increasing in size they become increasingly difficult to manage given

- their large code base,
- non-autonomous teams,
- lack in agility,
- difficult deployments and
- high commitment to specific technologies or even worse vendor lock-ins.

[2] Interaction Computing Execution Framework

Modern large-scale architectures, therefore, aim to find different possibilities in the field of SOA[3], and when advancing further also in Micro Services Architectures (MSAs). In particular, MSAs [9] claim to solve these problems by providing simple and easy-to-build applications at the expense of higher network load and more difficult system integration.

However, as mentioned above INTERLACE favours a different solution which has similar ideas but is in some of its most fundamental aspects very different; namely, the blockchain. Although the blockchain, like MSA, has a highly distributed nature, MSA is mainly an architectural stile whereas most blockchains are based on a data-centric approach and are quite specific in terms of scenarios of use. In addition, as discussed more fully in D2.2 [4], permissionless blockchain are also decentralised, which generally refers to a distribution of control in addition to a distribution of the computation. Although in this report we discussed the Hyperledger blockchain, which is permissioned, Hyperledger also has an interface to Ethereum, which is permissionless and which could be a useful addition to the Sardex/INTERLACE platform for example to record currency exchange transactions between circuits in different parts of the world that are pegged 1-1 to different fiat currencies. Finally, in the future an interface to Holochain, which is agent-centric and not data-centric, could also become relevant and useful [4, 5].

As discussed more fully in [4], desirable properties of blockchain technologies for INTERLACE are:

- Distribution
- Peer to Peer communication
- Immutable $\Rightarrow$ auditable
- Data storage
- Virtual Machine-like executions of code (chaincode or Smart Contracts)

Further, the use of Smart Contracts or chaincode (Hyperledger Fabric nomenclature) comes with various other attributes which are important to manage a mutual credit system. Code executions which run “on chain” are executed on every peer in the network. In order to be executed correctly a blockchain framework defines the roles of various system components, the state of the chain, and the code which is attached to a transaction and takes care of the business logic. The whole framework results in a workflow whose main steps are recorded and written to the blockchain.

Because the technology currently used by Sardex is mainly monolithic, it would be extremely difficult and risky to set up a complete distributed blockchain environment and swap systems from one day to the next. Rather, a roadmap to a completely distributed scenario is being planned, keeping in mind that the code base cannot be changed all at once. In fact, some of the functions of the original system are gradually being reimplemented as external microservices such as, for example, Search. Risk refers to the fact that the non-functional requirements of most blockchain frameworks are still evolving as the technology itself undergoes a continuous process of innovation. In addition, risk also refers to the continuous evolution and innovation that the business model of Sardex itself undergoes. Since blockchains are quite different application platforms which carry certain implications for how users should interact with the software product, any changes in the underlying framework or in the business/application layer need to be introduced gradually and carefully.

Consequently, the INTERLACE focus has been on understanding the complete core requirements, facilitating the AS(I)M approach, creating a generic model, and as much as possible remaining independent of the underlying technology used, in order to make the transition towards a fully

[3] Service-Oriented Architecture [6]

distributed, scalable, and reliable system clearer and with minimum risk. For INTERLACE this meant, further, that the agent-based ASIM methodology was used to specify a client-server model to create a Hyperledger Composer-based implementation.

The Hyperledger Composer business network is a blockchain architecture that also incorporates a Micro Services Architecture. Although this is not yet a fully distributed or decentralised scenario, it is scalable because new nodes can be added when extending to other circuits and the transaction processing load can be balanced between them. The permissioned Hyperledger architecture, therefore, allows better control of the blockchain functionalities and the enforcement of rules (business or otherwise), which would otherwise be very difficult or impossible to impose in a permissionless, public chain.

2.1.2 Testing

This subsection gives a quick overview of how testing may be processed for the prototype as well as for the final product. Detailed testing and a description of the testing activities will be reported in D4.1 and D4.2, which will be completed after the end of the project.

There are several levels of testing that need to take place. Since the requirements were defined in the ASIM language and were translated into an ICEF/coreASIM implementation, the test should take this intermediate implementation into account to verify the correctedness system. This, in fact, is integral to the iterative refinement process of the ASM methodology [2]. Traditional approaches like the v-model [7] usually test on four levels:

- Acceptance Testing
- System Testing
- Integration Testing
- Unit Testing

ASIM ground model testing cannot be put into one of these categories directly. Further, under the umbrella of modern software development methods like (large-scale) scrum, (scaled) agile different testing practices have been adopted besides the traditional approaches.

For D4.1/2 it is planned to test based on the results of the execution of the ASIM Models. Thus, each of the requirements has a specific output, which is then compared to the log of an execution produced by the new system. This is of course a pretty high-level way of testing. Compared to traditional testing it may be seen to lie between Integration and System Testing, but not properly in the realm of acceptance testing since the requirements defined by ASIMs are mathematically-based algorithms which are difficult to read and to verify by non-technical people.

Cucumber[4] may be used as a potential replacement tool for the ASIM comparison testing. Cucumber is a scenario-based testing tool which combines functional requirements into high-level system tests which are described as text as well as with a domain-specific language that enables the exectution of the test. A testing scenario may look like the one shown in Listing 2.1.

[4] https://cucumber.io

```
Feature: Perform Credit operations
  Move money from an account of member A to an account of member B

  Scenario: account B receives 100 Sardex from account A
    Given: Account A has a positive balance
      And: Account B is able to receive money
      And: Account A has a balance of 100
      And: Account B has a balance of 0
     When: Credit Transaction has performed successfully
     Then: account A has a balance of 0
      And: account B has a balance of 100
```

Listing 2.1: **Cucumber test example**

The various steps *Given* [initial context], *When* [event occurs], *Then* [ensure some outcomes] are based on a special language called Gherkin[5] whose elements can be connected to test implementations serving that scenario. Details can be found in the publicly available Cucumber documentation.

2.2 Solution Technologies

This last introductory part finally presents the planned solution stack which will be utilised for the prototypical implementation and is illustrated in Figure 2.1. Details on how these components work together are explained in depth in Section 3.

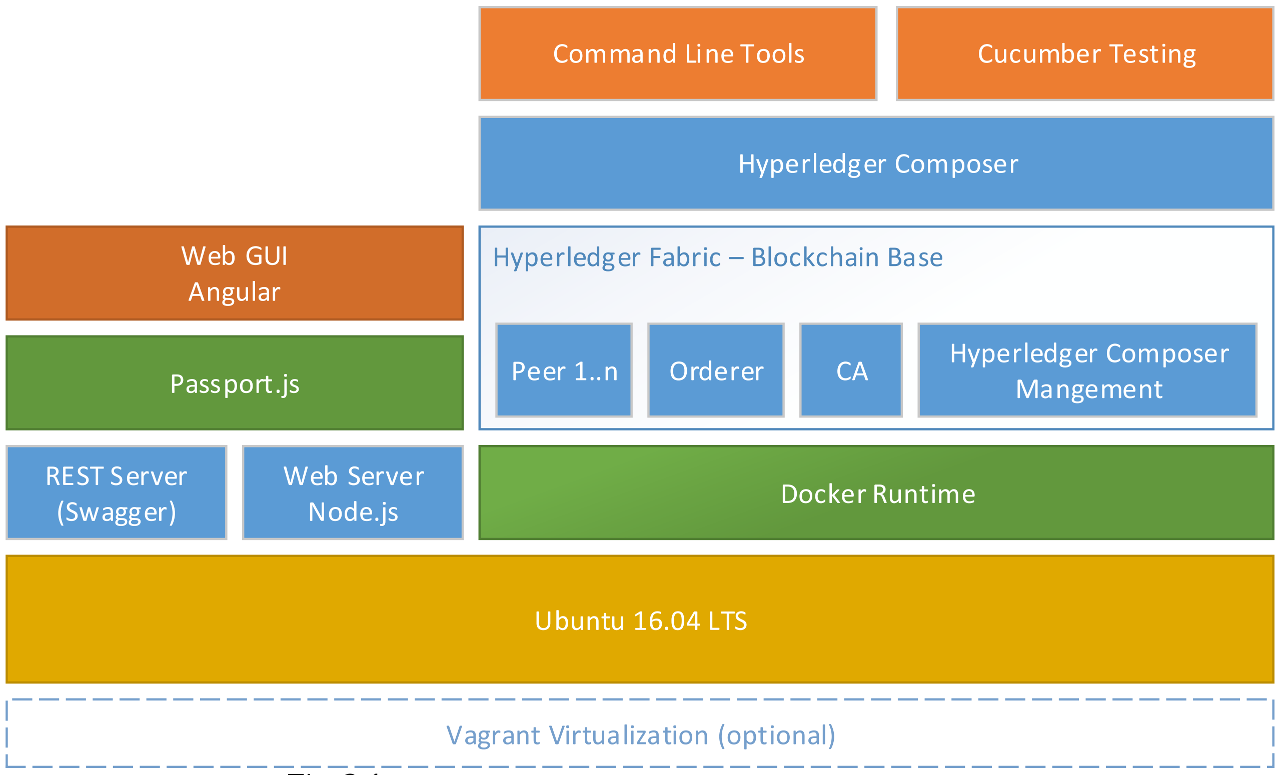


Fig. 2.1: **Technologies involved for the final solution stack**

Although detailed explanation will be given in Section 3, a simple structure introduction is given here.

[5] https://docs.cucumber.io/gherkin

The whole system currently runs on Ubuntu Version 16.04 LTS as Hyperledger Fabric together with Composer work properly only on the MacOs or Linux operating systems. Thus, optionally, a possibility exists which includes virtualizsation. On GitHub[6] a Vagrant-assisted Hyper-V or VirtualBox Machine may be started providing an Ubuntu 16.04 ecosystem, which is also explained in the next section.

On top of Ubuntu a Docker runtime hosts a Hyperledger Fabric docker-compose cluster which runs one to n peers, an Orderer, and a Certification Authority (CA), as well as managing a container which handles the Hyperledger Composer-specific part.

This base Fabric environment is managed by the Composer wrapper framework, which forms the key access port for the other components and provides help during implementation and the setting up of a business network.

Next, the Composer API provides stubs for the REST Server and the Web Server implementations, which really simplifies the programming necessary. These servers can be used to first load the application and connect it to the business network, and then execute various transactions/transfers. The Web Application is implemented using AngularJS and may be secured using Passport.js, which is an authentication middleware.

Finally, on top of the whole stack, Cucumber tests can be used to verify simple scenarios.

[6] https://github.com/hirsche/hyperledger

Chapter 3

Prototypical Implementation

Eduard Hirsch

3.1 Project Structure

Before introducing the actual architecture and implementation, the structure of the project is described in order to know what to find where. In Figure 3.1 the main directories are shown which also separate the main components of the business network.

```
Interlace Blockchain
├── chain
├── fabric
└── webapp
```

Fig. 3.1: **Project Directory Structure**

Explaining further, the folder *"chain"* contains a *Hyperledger Composer* implementation of a business network. This business network is the main implementation of the INTERLACE work to create a working blockchain that consists mainly of chaincode[7] implementations, but also of scripts to deploy and update the chaincode as well as to make the network accessible by starting it on the Hyperledger Fabric chain.

Hyperledger Composer aims to make blockchain application development easier by offering a large toolset together with a powerful framework. Its main purpose is to accelerate the time needed between requirements gathering and a final blockchain application, delivering fully functional business applications. Composer facilitates JavaScript and node.js in combination with proprietary language extensions to generate such business application and to run them on a Hyperledger Fabric instance.

"fabric" is the Hyperledger Fabric base blockchain needed for running a business network where the chaincode bits are executed on. The chaincode bits from directory *chain* are compiled to a *.bna* file ("blockchain network archive") which is then deployed to a Fabric network. These bna-files (also called banana files) can't be used by default by Hyperledger Fabric. They are, rather, part of a virtual infrastructure-like environment provided by Hyperledger Composer. Thus the API components of Composer provide wrappers to make a Composer implementation run on a pure Fabric network.

Finally a web application has been implemented which uses AngularJS, a front-end framework provided by Google, to work with the blockchain. This application is found in *webapp*. This application uses a Swagger[8]-based REST-server implementation which comes with the Hyperledger Composer framework.

A more detailed description of the directory structures can be found in subsequent sections where the respective parts are explained in depth.

[7] In the Hyperledger framework the language of Smart Contracts is referred to as 'chaincode'.
[8] `https://swagger.io/`

3.2 Architecture

The core architecture of the INTERLACE project presented next is multi-layered and concentrates on being scalable and highly distributable, in contrast to the current monolithic payment platform used by Sardex.

The chosen blockchain approach is going to be explained in detail. The technical challenges during the planning of that implementation strategies are discussed in subsequent sections of this chapter.

3.2.1 Sardex Network

In deliverables D2.1 and D3.1 the current architecture was specified using the AS(I)M approach and, as mentioned before, a working client-server application based on message passing was developed. This application was realized using the ICEF-framework which is founded on abstract state machines applying a programming language similar to ASM logic definitions.

The high-level functional model of D2.1 and D3.1 was used to derive a new payment network based on the blockchain approach. The new network platform involves using a publicly maintained and an easy-to-use blockchain which supports the principle of an interest-free mutual credit system enabling account-based balances rather than token-based (asset-based) currencies only. These considerations led to a prototypical implementation using Hyperledger Fabric together with Hyperledger Composer.

Independent of the choice of blockchain environment, it was also important to develop a central payment network which is not only reliable and verifiable but also (to a specific extent) controllable in order to impose the basic Sardex payment rules on it. Thus, the first implementation of the INTERLACE blockchain is actually centralised. The advantage is easy scalability to a distributed architecture as more circuits are added in other parts of Italy and beyond.

3.2.2 Hyperledger Fabric Network

The given network constraints were used while creating a new Hyperledger Fabric network which sets up a core environment offering a basic blockchain to interact with. Next, we discuss in some detail the actual implemented network along with a description for the various parts.

Figure 3.2 shows the current working environment, which uses the possibilities offered by Hyperledger Fabric. Upon closer scrutiny, Fabric actually imposes a structure on a newly created network, of which the following main components can be singled out for INTERLACE:

- Sardex as participating **organisation**
  - One **peer** (named: peer0.sardex.sardex.net)
  - Clients and Services connecting to the network
  - Sardex Membership Provider **(MSP)** with ID SardexMSP
- "Interlace" pseudo organisation
  - An **orderer** (named: orderer.sardex.net)
  - Interlace Membership Provider **(MSP)** with ID InterlaceOrdererMSP
- Certification Authority (**CA**)

Starting from a **CA**, a user may issue a unique identity which can be verified anytime by anyone participating in the network. These identities are part of the process of giving members of the circuit the right to work with and facilitate the network with particular roles and access privileges. The actual empowering of a user takes place inside of the membership provider (**MSB**).

However, the power of an MSP goes beyond simply keeping track of who is a network participant or member of a particular channel. An MSP is able to identify specific roles an actor might play within the scope of the organisation the MSP represents (e.g., admins, or as members of a sub-organisation group), and in general defines the foundation for giving access privileges in the context of a network and channel (e.g., channel admins, readers, writers). More details can be found in the documentation on the Hyperledger Fabric website.[9]

For the sake of simplicity and in order to set up the prototype network quickly, access/role management has been reduced to a minimum. Section 4.2 focuses on a more detailed explanation on how these aspects work or might be changed in case of larger-scale scenarios.

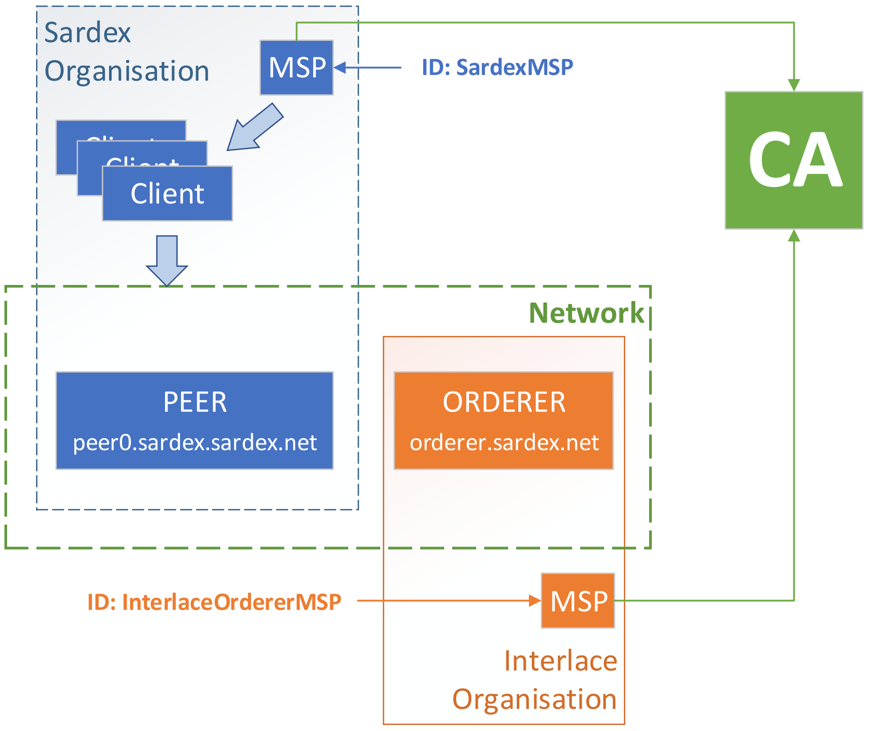


Fig. 3.2: **Network structure implemented by the Prototype**

The single **orderer** node in Figure 3.2 is responsible for atomic broadcasts, orders/batches transactions, and also signs each batch (block) to create unique and well-defined chains. The membership provider of the orderer uses a central and pre-configured certification authority **CA**.

This network configures and starts one **peer**. This peer can be called the core of the environment because its main task is to handle the so-called *smart logic* (chaincode). More specifically, peers maintain the ledger by first endorsing a transaction (e.g. credit and debit transfers in the case of INTERLACE), which they do by simulating it.[10] Then, in an intermediate step it is validated by the orderer and finally the peers commit it to their local ledger.

A very simple network is defined and described by the above scenario, which can be set up following the instructions of Section 3.3. Certainly, this scenario needs to be extended to a real-

[9] https://hyperledger-fabric.readthedocs.io/en/master/membership/membership.html
[10] This is the execution step in the Execute-Order-Validate Hyperledger architecture [1]

world environment, a few hints and considerations for which will be given in Figure 4.1 and explained in Section 4.3.

3.2.3 Network Configuration Files

There are three main configuration files providing a basic foundation for generating a pre-configured network. Those files are used to create the corresponding certificates as well as configuration files to spin up virtualised containers. Thus, the following three yaml files are an example of how to create a particular net as described in the previous section where the INTERLACE network was shown:

1. crypto-config.yaml
2. configfx.yaml
3. docker-compose.yaml

crypto-config.yaml

The *crypto-config.yaml* contains the network topology and therefore defines its basic structure. With the configuration file it is possible to use a tool called *cryptogen* which takes crypto-config.yaml as input to generate the cryptographic material necessary to run the blockchain. More specifically, *cryptogen* generates the keys for both the organisations and the components that belong to those organisations.

Listing 3.1 shows a part of crypto-config.ymal that defines one or more orderers. The yaml definition of the network peers is depicted in Listing 3.2.

```
1  OrdererOrgs:
2    # ---------------------------------------------------------------
3    # Orderer
4    # ---------------------------------------------------------------
5    - Name: InterlaceOrderer
6      Domain: sardex.net
7      # -------------------------------------------------------------
8      # "Specs" - See PeerOrgs below for complete description
9      # -------------------------------------------------------------
10     Specs:
11       - Hostname: orderer
```

Listing 3.1: **crypto-config.yaml excerpt – Orderer(s) definition**

Listing 3.1 shows that in the case of INTERLACE only one orderer is specified in line 5, with the name *InterlaceOrderer*. The network domain *sardex.net* is defined with the *Domain* key. Thus, together with the *Hostname* definition in line 11, the INTERLACE orderer can be reached using *orderer.sardex.net*.

Peers can be defined for the various organisations. For example, Listing 3.2 shows the definitions specific to the INTERLACE project. The first and only organisation for now is *Sardex* in the *sardex.net* domain. Similarly, Sardex gets the *Domain*-name *sardex.sardex.net*. Taking a look at the hierarchy one level down, peers will receive domain names like peer0.sardex.sardex.net or peer1.sardex.sardex.net. As the template *Count*-key only defines a value of *1*, there will only be one peer with sub-domain name *peer0*.

As mentioned above, user management is handled on a very small scale. Thus, when setting the user count to 0 in line 18 no users in addition to the administrator are defined.

```
 1 PeerOrgs:
 2   # ---------------------------------------------------------------
 3   # Org1
 4   # ---------------------------------------------------------------
 5   - Name: Sardex
 6     Domain: sardex.sardex.net
 7     EnableNodeOUs: true
 8     # Peer nodes and if applicable host name templates
 9     # for the newly created peers
10     Template:
11       Count: 1
12     # ---------------------------------------------------------------
13     # "Users"
14     # ---------------------------------------------------------------
15     # Count: The number of user accounts _in addition_ to Admin
16     # ---------------------------------------------------------------
17     Users:
18       Count: 0
```

Listing 3.2: **crypto-config.yaml excerpt - Peer(s) definition**

For more details on how to configure the network to a greater depth, kindly consult the official Hyperledger Fabric documentation,[11] which is a valuable source and should be studied in detail in order to build Fabric-based networks.

In addition to the original Fabric documentation a reference guide called "Hands-On Blockchain with Hyperledger" [10] was used to build this network.

configtx.yaml

The second configuration file is named *configtx.yaml* and contains different but also some redundant configuration bits for the blockchain network. Another tool called *configtxgen* picks up configtx.yaml and uses it to create configuration artefacts, thereby setting up a basic structure utilized by the actual blockchain network. These artefacts are:

- orderer *genesis block*
- channel *configuration transaction*
- an *anchor peer transaction* for each peer organisation

As stated in the Fabric documentation,[12]

> The orderer block is the Genesis Block for the ordering service, and the channel configuration transaction file is broadcast to the orderer at Channel creation time. The anchor peer transactions, as the name might suggest, specify each Organisation's Anchor Peer on this channel.

The core configuration excerpt can be seen in Listing 3.3. First, it defines an orderer genesis block called *InterlaceOrdererGenesis* containing one orderer handled by organization *Interlace* together with some capabilities (not discussed here) as well as a consortium using the network.

[11] https://hyperledger-fabric.readthedocs.io
[12] https://hyperledger-fabric.readthedocs.io/en/release-1.3/build_network.html

Second, it sets up a channel *InterlaceChannel* which is connected to a consortium named *InterlaceConsoritum*. This channel is used for credit and debit operations where currently only one organisation is defined: namely, *Sardex*.

Third, as mentioned initially, anchor peers are recorded into the ledger. This is done by submitting a transaction to the ledger which contains the main anchor peers. Anchor peers may be defined for an organization using the config-property *AnchorPeers*. The project defines a host called *"peer0.sardex.sardex.net"* that provides services at port *"7051"*, consistently with the crypto-config.yaml file. The host information is written into that initial transaction.

```
Profiles:
    InterlaceOrdererGenesis:
        Capabilities:
            <<: *ChannelCapabilities
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *Interlace
            Capabilities:
                <<: *OrdererCapabilities
        Consortiums:
            InterlaceConsortium:
                Organizations:
                    - *Sardex
    InterlaceChannel:
        Consortium: InterlaceConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Sardex
            Capabilities:
                <<: *ApplicationCapabilities
```

Listing 3.3: **configtx.yaml excerpt – Profiles definition**

docker-compose.yaml

Docker and Docker Compose are the core technologies used to start containerized services which finally start the actual blockchain and its components. Hyperledger Fabric developer offer images which are ready to be started right away when provided with the correct configuration locations.

The compose file defines four container images. In Listing 3.4 a shortened compose yaml-file shows the services (or images) started when *docker-compose up* is called. One for the certification authority (line 2), one for the orderer (line 4), one for the peer (line 6), and also an additional container (line 8) the peer is storing data to. That additional container is a NoSQL database called CouchDB.

```
1 services:
2   ca.sardex.sardex.net:
3     [...]
4   orderer.sardex.net:
5     [...]
6   peer0.sardex.sardex.net:
7     [...]
8   couchdb:
9     [...]
```

Listing 3.4: **docker-compose.yaml excerpt**

Also docker-Composer creates a virtual network environment where the service containers defined here can communicate between themselves using the particular domain names used in this configuration file.

Each of the services starts its respective application, which handles the allotted requests. The certification authority (CA) is started by calling *fabric-ca-server*, the orderer executes the eponymous command *orderer*, the only peer (peer0) runs *peer node start*, and the CouchDB image "couchdb" is started without specifying an additional command because the start-up process is handled by the container image itself.

The CA might be replaced for a real-world production system by a different authority supporting ECDSA certificates. Fabric only supplies this implementation to get a network quickly up and running.

Cryptographic Material

Except for CouchDB, all other services need to be configured with the network structure as well as the appropriate public-private key infrastructure. In the *chain* directory another folder called *network* can be found. This folder contains templates of configtx.ymal as well as crypto-config.yaml.

These template are, as mentioned, for generating the genesis block as well as the keys needed for the corresponding service. We have scripted the generation: it can be performed by calling *build.sh*, which is also located in the same directory as the yaml-files. The results of running the build script are

- interlace-channel.tx
- interlace-genesis.block
- crypto-config directory

These generated files as well as the directory are finally shared using docker volumes[13] for the respective services. A detailed description of how to configure the services can be found in the online documentation of Fabric[14] but also in [10].

3.3 Prototype

The prototype is a blockchain realisation of INTERLACE, can be found on GitHub[15] and is based on the specifications created in deliverable D3.1 [8] along with the ASIM specification of the requirements.

First, it is necessary to install the prerequisites which are available for Linux and Mac OS. Currently these are the recommended operating systems. However, with additional effort it might be possible to run the INTERLACE blockchain on Windows directly. To support Windows users a virtual machine set-up is also available.

Additionally, it is also important to set up a development environment described in the Composer GitHub repository. Even if development is not planned and setting up a complete environment not necessary, it is still advisable to install and start Composer Playground. Playground enables

[13] `https://docs.docker.com/storage/volumes/`
[14] `https://hyperledger-fabric.readthedocs.io`
[15] `https://github.com/InterlaceProject/InterlaceBlockchain`

someone to connect, alter, and test the INTERLACE payment network. Nevertheless, Playground is not required and it might be possible to use composer-cli or other methods to utilise the network.

3.3.1 Install

This part of the documents talks about how to set up and run the business network on your machine. However, before it is actually possible to begin it is necessary to install the pre-requisites which are listed at the Hyperledger Composer documentation.[16] The website also offers a download where a script for installing all the requirements for a machine is provided.

Nevertheless, for Windows users these scripts won't help because for now most of the packages are not yet prepared (as of this writing) for a Windows operating systems. We have prepared a virtual machine also for this user group, which also installs all the necessary frameworks and software tools during provisioning. This virtual machine is controlled by vagrant[17] and uses hyper-v or virtual box as hypervisor (two different branches). This VM configuration is published at GitHub[18].

Environment Start-Up

Once the Hyperledger environment is installed, the next step is to start the INTERLACE environment. To make communications uniform the blockchain is configured to publish all services under the host name "interlace.chain". To facilitate Windows users, in the suggested vagrant set-up the new hostname is added to your host-file at start-up time using the vagrant-hostmanager plug-in. Thus there is no need to configure the name manually.

Configure Hostnames

For non-vagrant users it is important before executing the local set-up to add a host name entry for "interlace.chain". Usually this entry will point to IP 127.0.0.1 (localhost). On a production system or if it was chosen to start the Hyperledger Composer services on a public interface, the IP needs to be fixed accordingly. Here is a list of host file locations according to different operating systems types:

- Mac OS: /private/etc/hosts
- Linux: /etc/hosts
- Windows: C:\Windows\System32\drivers\etc\hosts

The format may vary a little but usually a new host with its hostname is defined using its IP and the desired host name as

```
127.0.0.1        interlace.chain
```

Depending on the operating system, it might be also necessary to update and restart the corresponding services (e.g. MacOS).

Run the Fabric blockchain (the first time)

[16] https://hyperledger.github.io/composer/latest/installing/installing-prereqs.html
[17] https://www.vagrantup.com/
[18] https://github.com/hirsche/hyperledger

At this stage, the main configurations have been performed and Hyperledger Fabric can be started, which acts as a base for Hyperledger Composer. To continue, if not yet done the GitHub repository[19] needs to be downloaded by using the git visioning system by calling:

```
git clone https://github.com/InterlaceProject/InterlaceBlockchain.git
```

In the directory "InterlaceBlockchain" that is created the business network implementation, including a web application, can be found. The next listing shows the bash script that downloads several Fabric docker containers and finally starts them using docker-composer:[20]

```
cd fabric
./downloadFabric.sh # updates images - only the first time necessary
./startFabric.sh # start up docker environment using docker-compose
```

Initialize Interlace-Chain

Finally, after Fabric has been started the next step is to initialize the blockchain with

```
cd chain
./initNetwork.sh # use Hyperledger Composer to create a business network and
  ↪ deploy it
```

./initNetwork.sh will copy all models and script to the network peers to make them accessible in the Hyperledger blockchain. The last step in the script starts the business network.

It may be more convenient to access the network and test CreditTransfer or DebitTransfer transactions using Playground. **data.json** should act as a helper to initialise the network by hand, but it is recommended to update the JavaScript function *initBlockchain(transfer)* in *./chain/lib/init.js*. That chaincode part is executed when transaction InitBlockchain is submitted. Be careful to run **InitBlockchain** only once otherwise errors or duplicate entries might happen resulting in an inconsistent chain.

Network updates after chaincode changes

After changes to the acl, cto, queries, the libraries, or other parts of the core chaincode application, the network needs to be updated. This can be achieved by executing

```
cd chain
./updateNetwork.sh
```

This script reads the current version number of the *package.json* file, increments it by one, and creates a new bna package. If the scripts are correct and the bna-package can be created it is deployed to the peers and the network is updated to a new, higher network version which will utilize the new bna package.

Shutting down

Sometimes it is useful to throw away everything and restart from scratch. To tear down Fabric and remove card left-overs execute:

```
cd fabric
./teardownFabric.sh
./deletePlaygroundCards.sh
```

[19] https://github.com/InterlaceProject/InterlaceBlockchain
[20] https://docs.docker.com/compose/

Start a REST server

Once the network is running (no Playground needed) it is also possible to start an HTTP Server which allows to interact with the network over REST. The script

```
cd chain
./startRestServer.sh
```

starts the server and provides GUI access to the RESTful interface by opening

`http://interlace.chain:3000/explorer`

in a browser. When accessing the REST interface from an external application, it may be reached over

`http://interlace.chain:3000/`

If the host interlace.chain has not been set up and all the services are running locally without a VM, it might be also possible to use localhost instead of interlace.chain as host name. Nevertheless, it is highly recommended to use "interalce.chain" because everything has been tested using that particular host name.

3.3.2 Working with the environment

Next, a closer look is taken on how the environment might be facilitated using different approaches. It is possible to connect to the chain using composer-cli, taking advantage of Composer Playground (the graphical interface) or using the simple web front-end created for the project.

Start and test network with Playground

If you've decided to install and use Composer Playground it can be started using this command

```
composer-playground
```

The standard configuration opens a browser connecting to Playground at localhost with port 8080. If you are running Playground in a separate virtual environment like e.g. in a docker container, it may be necessary to start the browser manually, determine the VM/Containers IPs, and fill in the address manually in the URL field.

The Admin Cards

Composer Playground helps by providing a basic web interface to interact with the Hyperledger Fabric blockchain, on top of which Hyperledger Composer acts as an additional wrapper. Composer creates cards in order to connect to the blockchain. These cards can be created over the Playground web interface or over the command line interface. For INTERLACE these cards are created by *initNetwork.sh*, which is explained in Section 3.3. When these scripts are executed, and no error messages are issued, two cards should have been created and should become visible when opening Hyperledger Playground. This is illustrated in Figure 3.3.

These cards provide all the information needed to connect to the INTERLACE blockchain business network. In particular, in the present case two cards are installed:

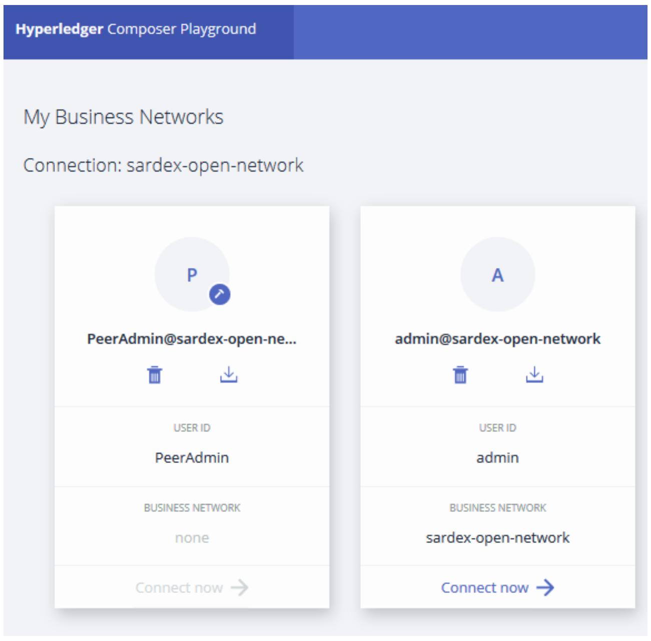


Fig. 3.3: **Admin Cards in Hyperledger Playground**

- Peer Admin Card (PeerAdmin@sardex-open-network)
- Business Network Card (admin@sardex-open-network)

Peer Admin Cards are cards (as the name suggests) used to interact with the peers. Users connecting over that card receive the permissions to manage chaincode deployments or changes on the peers. Thus they are a crucial part of every network.

Access to Interlace/Sardex Business Network is granted to another user through the provision of a *Business Network Card* which is called *admin@sardex-open-network*.

For INTERLACE we currently only have one user in place, the admin user. However, if the network gets deployed it will become necessary to grant access to other users, which can be done by creating an additional Business Network Card for each new user. Be aware that for each new user a participant needs to be registered first in order to link it with a new network card.

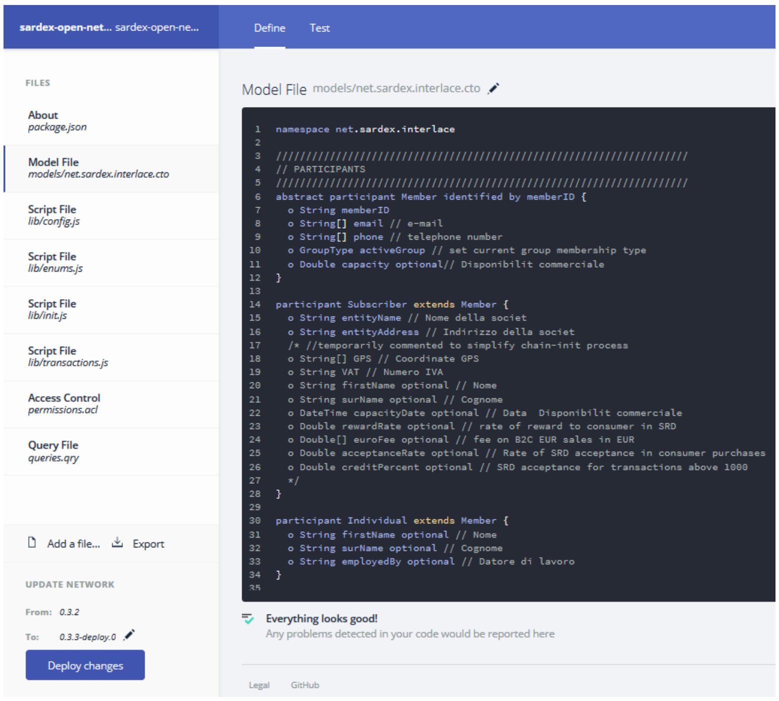
Edit Network

Figure 3.4 illustrates how a particular business network may be edited over Playground directly. For INTERLACE this means that you can e.g. quickly try out some changes on the JavaScript files and see if the changes are deployable by pressing the "Deploy changes" button. However, most developers might prefer using a common IDE,[21] which offers far better assistance during development, and use the scripts *initNetwork.sh* and *updateNetwork.sh* provided.

Test Network

The INTERLACE blockchain might be also tested directly with Playground web interface instead of using the Composer cli tools or launching the additional web application provided. Figure 3.5 shows the Playground test environment that comes with Playground and is started by pressing "Test" in the menu bar.

[21] https://en.wikipedia.org/wiki/Integrated_development_environment



```
namespace net.sardex.interlace

////////////////////////////////////////////////////////////////////
// PARTICIPANTS
////////////////////////////////////////////////////////////////////
abstract participant Member identified by memberID {
  o String memberID
  o String[] email // e-mail
  o String[] phone // telephone number
  o GroupType activeGroup // set current group membership type
  o Double capacity optional// Disponibilit commerciale
}

participant Subscriber extends Member {
  o String entityName // Nome della societ
  o String entityAddress // Indirizzo della societ
  /* //temporarily commented to simplify chain-init process
  o String[] GPS // Coordinate GPS
  o String VAT // Numero IVA
  o String firstName optional // Nome
  o String surName optional // Cognome
  o DateTime capacityDate optional // Data  Disponibilit commerciale
  o Double rewardRate optional // rate of reward to consumer in SRD
  o Double[] euroFee optional // fee on B2C EUR sales in EUR
  o Double acceptanceRate optional // Rate of SRD acceptance in consumer purchases
  o Double creditPercent optional // SRD acceptance for transactions above 1000
  */
}

participant Individual extends Member {
  o String firstName optional // Nome
  o String surName optional // Cognome
  o String employedBy optional // Datore di lavoro
}

```

Everything looks good!
Any problems detected in your code would be reported here

Fig. 3.4: **Edit a network in Hyperledger Playground**

On the left bar of that view you are provided with a list of the INTERLACE participants, assets, as well as entry call "All Transactions". As the menu names might suggest, when clicking on them a list of those items which have been created on the chain is displayed. E.g., this view in 3.5 lists all *Individuals* taking part in the Interlace-Test network. As shown in the main area of the screenshot, two "Individual" participants have been registered. One member with ID "m1" and one with ID "m2".

In addition, it is possible here to add and change entries for participants and assets you might change for customised tests.

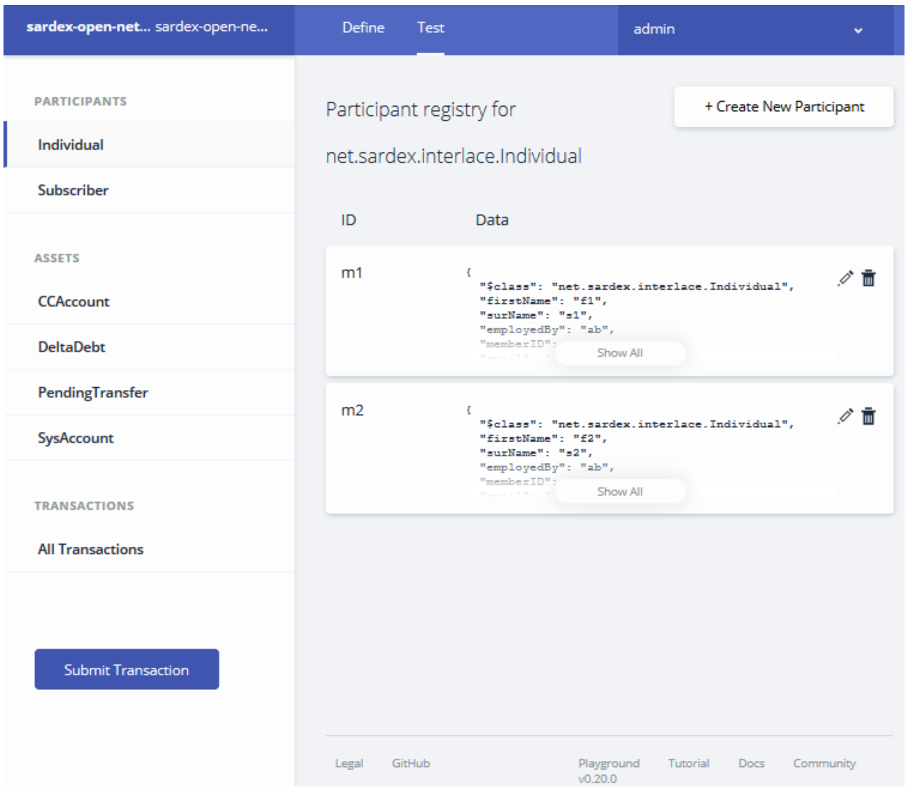
Finally, the "All Transactions" menu entry guides to a list of all transactions executed on the INTERLACE chain. This list contains of course not just the transactions somebody has submitted but also entries like e.g. "AddParticipant" or "IssueIdentity". Thus, all changes to the blockchain are recorded and can be found here.

Submit a transaction

Figure 3.6 shows a submission dialog, which opens when the "Submit Transaction" button in screenshot 3.5 is pressed. This dialog gives the possibility to select one of all possible transactions executable on the INTERLACE network. In this dialog a *CreditTransfer* has been selected.

In the black text-box the properties of that transactions can be provided as a JSON-String. The interface provides default transaction-specific values.

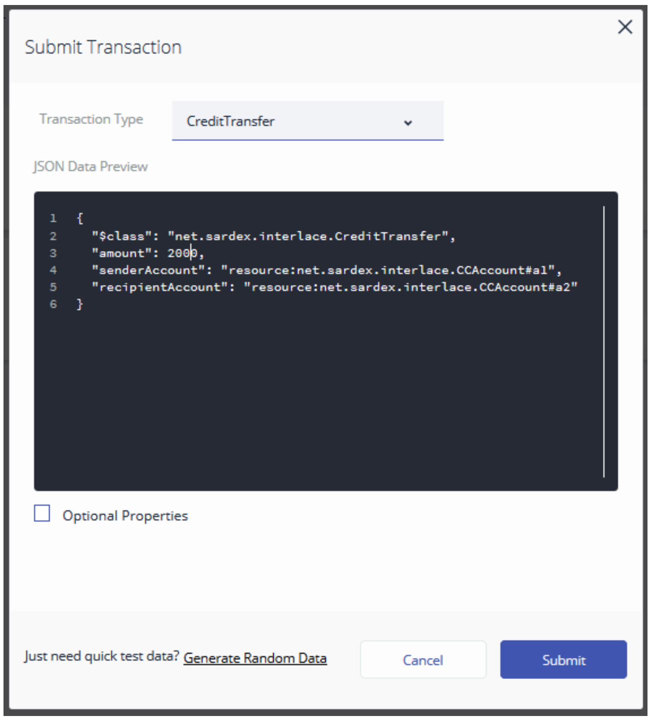
Once all the necessary properties have been provided, pressing "Submit" tries to commit the transaction to the blockchain. If an error is encountered, it is shown in red font attached to the same dialog. When everything goes as planned the transaction is endorsed, ordered, and committed to the peers. The execution results will finally show up in the JSON records used for the assets and participants as well as in the transaction log of the chain.



Participant registry for net.sardex.interlace.Individual

ID	Data
m1	`{ "$class": "net.sardex.interlace.Individual", "firstName": "f1", "surName": "s1", "employedBy": "ab", "memberID":`
m2	`{ "$class": "net.sardex.interlace.Individual", "firstName": "f2", "surName": "s2", "employedBy": "ab", "memberID":`

Fig. 3.5: **Test a network in Hyperledger Playground**



Transaction Type: CreditTransfer

JSON Data Preview

```
{
  "$class": "net.sardex.interlace.CreditTransfer",
  "amount": 2000,
  "senderAccount": "resource:net.sardex.interlace.CCAccount#a1",
  "recipientAccount": "resource:net.sardex.interlace.CCAccount#a2"
}
```

Fig. 3.6: **Submit a (credit) transfer in Hyperledger Playground**

Details on how to configure and initialize INTERLACE transactions are covered in Section 3.4.2 which discusses the technical details.

Run Transactions with composer-cli

Initialise network transaction:

```
composer transaction submit -c admin@sardex-open-network -d  '{ "$class": "net.
  ↪ sardex.interlace.InitBlockchain" }'
```

The InitBlockchain transaction sets up some basic accounts as well as demo members that can issue simple transactions right away.

Submit a credit transfer from account a1 to a2 with amount of 800 SRD:

```
composer transaction submit -c admin@sardex-open-network -d  '{ "$class": "net.
  ↪ sardex.interlace.CreditTransfer", "amount": 800, "fromAccount": "resource:
  ↪ net.sardex.interlace.CCAccount#a1", "toAccount": "resource:net.sardex.
  ↪ interlace.CCAccount#a2" }'
```

Submit a debit transfer from account a1 to a2 with amount of 200 SRD:

```
composer transaction submit -c admin@sardex-open-network -d  '{ "$class": "net.
  ↪ sardex.interlace.DebitTransfer", "amount": 200, "fromAccount": "resource:
  ↪ net.sardex.interlace.CCAccount#a1", "toAccount": "resource:net.sardex.
  ↪ interlace.CCAccount#a2" }'
```

A successful debit transfer creates a PendingTransfer entry with status Pending containing an OTP (one-time password). This OTP can be used by the debitor to confirm the transaction. Thus, in the next example "995317396" is used to call a transaction DebitTransferAcknowledge to acknowledge the debit transfer:

```
composer transaction submit -c admin@sardex-open-network -d  '{ "$class": "net.
  ↪ sardex.interlace.DebitTransferAcknowledge", "transfer": "resource:net.
  ↪ sardex.interlace.PendingTransfer#995317396" }'
```

The web front-end

The web front-end currently is a simple website generated by a Yeoman generator provided by the Composer community. The web application can be found in the web app directory.

In order to get the web application to run properly it is necessary to start up the whole network and start the REST server as described in the previous steps.

The web app is based on AngularJS and needs various node.js packages downloaded and installed, which is achieved by calling

```
cd webapp
npm install
```

After that a development server can be started by calling

```
cd webapp
npm start
```

npm will start a web server at port 4200. If you work locally it also tries to open a browser which shows the web application. Otherwise one needs to start a browser manually and enter the URL. This is the URL where the server can be reached:

`http://interlace.chain:4200`

The webpage is based on AngularJS and communicates over REST with our previously started REST server, enabling asynchronous AJAX-request.

3.4 The Chaincode

This section describes the core business logic, which is found in the *chain* directory. The shell scripts ending with *.sh* were discussed in Section 3.3, except for *startRestServer.sh* which is handled in the technical details Section 3.5. This section focuses on the chaincode implementation which is stored in the directory *lib* illustrated in Figure 3.7.

```
chain
├── lib/
│   ├── config.js
│   ├── enums.js
│   ├── init.js
│   └── transactions.js
├── models/
├── network/
├── connection.json
├── data.json
├── initNetwork.sh
├── package-lock.json
├── package.json
├── permissions.acl
├── queries.qry
├── updateNetwork.sh
└── startRestServer.sh
```

Fig. 3.7: **Chaincode Directory Structure**

In the lib directory, *config.js* contains the main configurations including things like time-outs, quick-transfer amounts, and transfer types and account type mappings, which are done in the form of a JSON object. Unfortunately, Composer doesn't offer static access to enumeration constants. All enumeration values of a type need to be addressed by a string value. This approach is quite dangerous and gives potentially space for a lot of common errors. INTERLACE tries to solve this issue by setting up an object with "frozen"[22] attributes.

In Listing 3.5 a JavaScript mapping of the cto model enum type *Unit* in 3.6 is illustrated.

[22] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

```
var Unit = Object.freeze({
  ’Euro’: ’Euro’,
  ’SRD’: ’SRD’
});
```

Listing 3.5: **JavaScript enumeration mapping**

Using this way of enum-mapping, a proper development environment suggests and auto-completes the possible values of an e.g. Unit when the user works with it. Instead, there is the possibility, as mentioned, to directly work with string values. But writing *Unit.Euro* instead of *"Euro"* ensures during development when sources are linted[23] that the correct enumeration names have been applied.

```
enum Unit {
  o Euro
  o SRD
}
```

Listing 3.6: **enum in CTO-model**

3.4.1 Linking Transactions

There are two JavaScript files which contain the available transactions of the network: *init.js* and *transactions.js*. Before going into the details of what these files contain, we give a simple example of how transactions of the CTO model are mapped to a JavaScript function. Listing 3.7 gives a short example of the syntax through which *CreditTransfer* transactions are linked to a JavaScript function with the same name.

```
1  /**
2   * CreditTransfer transaction
3   * @param {net.sardex.interlace.CreditTransfer} transfer
4   * @transaction
5   */
6  async function CreditTransfer(transfer) { [...] }
```

Listing 3.7: **Connection of JavaScript function CreditTransfer to CTO-model transaction type**

In this example the JavaScript name of the function is not important. Rather, what matters is the notation *@transaction*, as well as the definition of the parameter(s) of the function and of their type. More specifically, on line 3 the parameter *transfer* is linked to the CTO model type *net.sardex.interlace.CreditTransfer*. When a CreditTransfer is invoked, usually using a JSON string, this string is converted to a type in JavaScript that has the same properties as *net.sardex.interlace.CreditTransfer* and that is provided as *transfer* input parameter.

Then, during execution the *transfer* input parameter may be used like any JavaScript object and contains all the information necessary to process that transaction request. In JS, it is important that the code be deterministic and that it evaluates to the same result on different peers.

[23] https://en.wikipedia.org/wiki/Lint_%28software%29

3.4.2 Init Blockchain

The only transaction in *init.js* should be executed only once, otherwise it will lead to an inconsistent blockchain if executed twice. The reason is that this initialisation script sets up a couple of participants with an account asset each to immediately test the blockchain. Thus, after the "InitBlockchain" transaction has been executed the chain contains at least two members and two accounts in order to move money around by applying credit and debit operations. Therefore, executing the initialisation multiple times will create a pair of participants each time, leading to a configuration that does not match other parts of the model.

Listing 3.8 shows parts of the actual creation of an asset as well as of a participant. It uses *getFactory()*, which is part of the Composer API to generate new instances of various types. The factory offers a function *newResource* to actually create first an "Individual" from namespace *net.sardex.interlace* and afterwards a "CCAccount" residing in the same namespace. The namespace is not visible in the script of Listing 3.8 because it has been configured in the config.js file and applied to the config object.

Function *newResource* uses namespace, type name, and identifier as parameters and, in the case of line 3, for example, gives back a JavaScript representation of "net.sardex.interlace.Individual". The same principle applies to line 8 when a CCAccount is created.

To write a new resource to the ledger a registry of the type super-category has to be acquired.

Given that the Individual is a participant, the *getParticipantRegistry* function needs to be called in order to write it to the chain. For any asset like the CCAccount, it is necessary to call the *getAssetRegistry* function to receive the right registry.

```
1  let factory = getFactory();
2
3  let m1 = factory.newResource(config.NS, 'Individual', 'm1');
4  m1.firstName='f1';
5  [...]
6
7  let a1 = factory.newResource(config.NS, 'CCAccount', 'a1');
8  [...]
9  a1.balance=1000;
10 a1.member=factory.newRelationship(config.NS, 'Individual', 'm1');
11 [...]
12
13 let partReg = await getParticipantRegistry(config.NS + '.Individual');
14 await partReg.addAll([m1]);
15
16 let accReg = await getAssetRegistry(config.NS + '.CCAccount');
17 await accReg.addAll([a1]);
```

Listing 3.8: **Chaincode adding a new resource in *initBlockchain* function**

Once the registries are available, they can be called as in lines 14 and 17 with the matching type-category, to finally ask to add a new entry to the chain. The Hyperledger Composer API also has functions for reading, removing or update included. The documentation for the AssetRegistry[24] and for the *ParticipantRegistry*[25] can be found in the Composer documentation.

[24] https://hyperledger.github.io/composer/v0.19/api/runtime-assetregistry
[25] https://hyperledger.github.io/composer/v0.19/api/runtime-participantregistry

3.4.3 Main Payment Transactions

The current implementation comes with credit and debit transactions that work according to the specifications made in D2.1 [3] and D3.1 [8], except for the requirement concerning the tracking of the debt position over time, *DeltaDebt*, which can be found in D2.3 [5].

CreditTransfer chaincode

Let's now focus on the first of the two payment transactions: the credit transfer. Listing 3.9 shows the core of the JavaScript function without the additional function wrappers.

The code was written in a way that is supposed to be readable by people who may not be JavaScript experts. Nevertheless, the keyword *await* might need a brief explanation:[26] *await* deals with asynchronous JavaScript function calls and (in simplified terms) just waits until the prefixed function completes its task. If *await* were missing, an asynchronous JavaScript (like previewCheck) would be called and executed in the background and the execution of the current thread would continue immediately.

```
//some basic checks
await checkAmountPlausible(transfer);

// preview check throws error in case of violation
await previewCheck(transfer);

// account limits checks throws error in case of violation
await accountLimitCheck(
  transfer.fromAccount,
  transfer.toAccount,
  transfer.amount);

// check account limits and emits event if violated
await checkAccountLimitsAlerts(transfer.fromAccount);

// perform the transfer
await moveMoney(transfer);
```

Listing 3.9: **CreditTransfer JavaScript**

The *transfer* object used in the listing is pre-filled by the chaincode API after a CreditTransfer is invoked. It is created from the parameters and values of the JSON object that is part of the submitted transaction and that follows the structure of the CTO model of type *net.sardex.interlace.CreditTransfer*.

DebitTransfer chaincode

The debit operation, shown in Listing 3.10, is a bit more complex compared to a basic credit operation. In that listing only the important part of the function is shown. The leading part calling *checkAmountPlausible*, *previewCheck*, and *accountLimitCheck* is exactly the same as in CreditTransfer and, therefore, it was left out to increase readability.

One note for debit operations resulting from the specifications is that the owner of the fromAccount from the debit transfer is the debitor, who is also the buyer. Whereas the toAccount

[26] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

owner is regarded as the creditor who is selling something. Please see Figure 2.7 in D3.1 [8] for the definition of fromAccount, toAccount, and related concepts.

Continuing further with the implementation, the core of the debit operation checks if an immediate transfer is possible (amount smaller than a predefined value) or if a confirmation of the other party is necessary. In case of an immediate transfer the money is moved the same way as in CreditTransfer by calling the *moveMoney* function. If the transfer amount is above a threshold (currently 100 SRD), however, a confirmation needs to be obtained from the debitor who is in our case the owner of the fromAccount of the transfer.

```javascript
// check for immediate transfer possibility
if (transfer.amount <= config.debit.quick_transfer_amount) {
  // perform the transfer
  await moveMoney(transfer);

  // check account limits and emits event if violated
  await checkAccountLimitsAlerts(transfer.fromAccount);
} else { // requires confirmation
  // add the debit transfer to the pending queue
  let otp = await insertPendingTransfer(transfer);

  // create confirmation event RequestDebitAckReqAnswCompletion
  // which is including the OTP and the sender account (of debitor)
  [...]

  // emit the event
  emit(confirmReq);
}
```

Listing 3.10: **DebitTransfer JavaScript**

If a confirmation is needed the transaction *DebitTransfer* cannot move the amount from account A to account B instantaneously. To remember all values of that transaction they are stored inside asset *PendingTransfer* which uses an OTP as identifier. The insertion done by Function *insertPendingTransfer* returns a new OTP when successfully, creating a transfer which is awaiting confirmation.

In order to tell a user that there is a pending confirmation waiting DebitTransfer issues an event called *RequestDebitAcknowledge* of namespace *net.sardex.interlace* defined in the CTO model. When instantiated, the event obtains a reference to the just created *PendingTransfer* which is done by using the *factory* object (received by the composer API) and calling *newRelationship* which utilizes the OTP references.

DebitTransferAcknowledge chaincode

Each entry in PendingTransfer is identified by an OTP in a way that a transfer can be unambiguously selected by providing an OTP. Currently the OTP is generated by hashing the transaction id of the debit transfer. However, if a debitor would like to confirm a transfer of which he was informed by the emitted event, he just needs to pass the OTP as property for the *DebitTransferAcknowledge* transaction.

Note: For production systems this identification might include an additional key since for large numbers of PendingTransfers the OTP generated in this way might not be unique anymore.

Listing 3.11 shows the main part of *DebitTransferAcknowledge*, which is different from the other transactions. The variable *ack* in the script is a transaction object provided to the function which contains the *PendingTransfer* read into variable *pT*. Then the code tries to verify the *TransactionStatus*. If the transaction is in state *Pending* it may proceed; in any other case an error is thrown and the transaction is discarded (!not! recorded into the ledger).

If the process is continued the timestamp of the transaction is checked against the *expires* date property of the pending transfer. Consequently, if expired, the state is updated to *"Expired"* and function *DebitTransferAcknowledge* returns with the corresponding status *AcknowledgeStatus*.

```
 1  [..]
 2  //get pending transaction
 3  let pT = ack.transfer;
 4
 5  // verify state of pending transfer
 6  if (pT.state !== TransactionStatus.Pending) {
 7    throw new Error(’Transfer is not in state "’ +
 8      TransactionStatus.Pending + ’" but in state "’ + pT.state + ’"’);
 9  }
10
11  // varify if pending transaction has been expired
12  if (ack.timestamp >= pT.expires) {
13    //update state from Pending to Rejected
14    await updatePendingTransaction(pT, TransactionStatus.Expired);
15
16    //prepare return message
17    rS.status = TransactionStatus.Expired;
18    rS.description = ’OTP ’ + pT.otp + ’ is expired.’;
19    return rS; //TODO: raise event
20  }
21  [..]
```

Listing 3.11: **RequestDebitAcknowledge JavaScript excerpt**

In the later parts of the function, after all of the checks have been passed and the debit transaction can be performed, the execution steps are pretty much the same as in a Credit-Transfer. The only difference is that in case of success or error the PendingTransfer asset of the transaction needs to be updated to Performed or Rejected, respectively. A function called *updatePendingTransaction*, also used in line 14 of Listing 3.11, takes care of putting the asset PendingTransfer into the right state.

Additionally, in all cases in which *updatePendingTransaction* has been applied, also a status object called *AcknowledgeStatus* is created and returned by the function that contains the resulting status, along with an error message if something went wrong.

moveMoney & DeltaDebt

Actual movement of an amount from one to an other account is performed by basic addition and subtraction applied to the balances of the respective accounts and, finally, by updating the account assets in the ledger by means of the registry provided by the Hyperledger Composer API.

Another important step, however, which is part of this money transfer and is shown in Listing 3.12, is an additional functionality which was specified in D2.3 [5]. This logic collects all debts, thus, every transactions which are causing the balance to go negative or to increase (in absolute value) an already negative balance. These debts are collected because they are handled similarly to a loan. ALthough they don't incur interest or any additional fees, these debts have a due date of 12 months by when they need to be paid back. See the Appendix of [5] for more details.

```
[..]
// check balance if DeltaDebt entry needs to be added
// !after amount has been substracted!
if (transfer.fromAccount.balance < 0) await createDeltaDebt(transfer);
// check balance if clearing an open DeltaDebt is necessary
// !before amount has been added!
if ((transfer.toAccount.balance - transfer.amount) < 0) {
  await clearDebt(transfer);
}
[..]
```

Listing 3.12: **moveMoney JavaScript excerpt**

The creation of a debt handled by *createDeltaDebt* is straightforward and just adds an entry to asset *DeltaDebt* if a transaction has been detected which causes a negative balance or makes an already negative balance more negative. A new *DeltaDebt* entry receives an original amount, a current amount, an owner id, and of course a due date by when it needs to be paid back.

If however a transaction adds money to the account owner's balance, say with a positive amount "Amount", former *DeltaDebt* entries may be cleared, which is done by querying all debts with a still unpaid amount. This is illustrated in Listing 3.13, where a rich query called *selectDeltaDebt* is used to account for all of those unpaid debts. Details of this and other queries are discussed in subsection 3.4.5.

```
// query result sorted by "oldest" first
let openDelta =
  await query('selectDeltaDebt', {ID: (transfer.toAccount.member.memberID)});
```

Listing 3.13: **clearDebt JavaScript excerpt**

All selected debts are iterated, starting with the oldest first. The available portion of Amount gets subtracted from the current *DeltaDebt* asset. If Amount is bigger than the current debt being examined, the rest of Amount ($amount = amount - deptPos$) is used in the next iteration step towards the clearing of the next-oldest entry of *DeltaDebt*. Thus, this loop continues either until Amount has been all used up or all debts have been paid back, which also means in the latter case that the balance goes back to having a zero or positive value.

3.4.4 Additional Transactions

There is an additional transaction important for maintenance issues. This transaction, called *CleanupPendingTransfer*, is illustrated in Listing 3.14 and takes care of handling old entries of asset *PendingTransfer*.

In some cases pending transfers stay unconfirmed for ever because they e.g. were issued wrongly by a user, or were duplicates if a connection had been interrupted together with a lost emitted acknowledge-request event. In such cases, for legal and security reasons these transfers have to be set to *Expired*.

```
let expiredPending =
  await query('selectExpiredPendingTransfers', {now: (transfer.timestamp)});
let aR = await getAssetRegistry(config.NS + '.PendingTransfer');

// change all states to expired
expiredPending.forEach(p => p.state = TransactionStatus.Expired);
await aR.updateAll(expiredPending);
```

Listing 3.14: **clearDebt JavaScript excerpt**

First, the chaincode shown selects all expired entries with query *selectExpiredPendingTransfers*. Then it iterates all of them applying the new state. Finally, all entries are updated using the *AssetRegistry*.

As this is a maintenance transaction it should be executed only by an admin or a user with a specific maintenance role.

3.4.5 Queries

Using Hyperledger Composer it is easy to read data written into the ledger. This can be done with its bespoke query language by defining a file called *queries.qry*.

The Hyperledger Composer Query Language[27] is used in INTERLACE to read information about *PendingTransfer* as well as about *DeltaDebt*. Although it looks similar to SQL, it has only a very limited set of operators. Nevertheless, for INTERLACE these limitations are not relevant because simple selection and filtering is sufficient, as can be observed in Listing 3.15.

The first query in the listing *selectExpiredPendingTransfers* is used by transaction *CleanupPendingTransfers* to find all transfers which are outdated and should be marked as *Expired*. More specifically, the *SELECT* keyword expects an asset defined in the CTO model and will return all the entries in the ledger filtered by the *WHERE* condition.

```
query selectExpiredPendingTransfers {
  description: "select all expired transfer which are still in state pending"
  statement:
      SELECT net.sardex.interlace.PendingTransfer
       WHERE ((expires <= _$now) AND (state == 'Pending'))
}
query selectDeltaDebt {
  description: "select all open debts"
  statement:
      SELECT net.sardex.interlace.DeltaDebt
       WHERE ((deptPos > 0) AND (debitorID == _$ID))
       ORDER BY [created ASC]
}
```

Listing 3.15: **INTERLACE business network queries**

Here we filter against property *expires* and state equals to 'Pending'. Parameters like *_$now* may be supplied using a JSON-like reference from JavaScript.[28]

[27] https://hyperledger.github.io/composer/v0.19/reference/query-language
[28] https://hyperledger.github.io/composer/v0.19/api/client-businessnetworkconnection#buildquery

In query *selectDeltaDebt*, all the open debts of a particular debitor (member of the circuit) who has id *_$ID* are selected. Like parameter *_$now*, parameter *_$ID* needs to be set by the calling counterparty.

3.4.6 Access Control Language File

The INTERLACE prototype currently only has one user, who is also the admin and user of the whole network – no certificates have been issued for other users. This can be easily changed by adopting the .acl file of the implementation and binding certificates to participants. However, to reduce complexity, especially for people who work first with the demo implementation, the set-up has been kept simple in this regard.

Listing 3.16 shows the INTERLACE Access Control Language file, permissons.acl, which grants access to any user with any operation available.

```
rule Default {
    description: "Allow all participants access to all resources"
    participant: "ANY"
    operation: ALL
    resource: "net.sardex.interlace.*"
    action: ALLOW
}

rule SystemACL {
  description:  "System ACL to permit all access"
  participant: "ANY"
  operation: ALL
  resource: "org.hyperledger.composer.system.**"
  action: ALLOW
}
```

Listing 3.16: **Access control configuration for INTERLACE**

Details on how to refine access can be found in the Hyperledger Composer documentation.[29]

3.4.7 Deployment

As mentioned in Section 3.3.1, the deployment of the chaincode application has to happen in several steps which are handled by *initNetwork.sh*, or after consecutive changes with *updateNetwork.sh*. Both scripts utilize the hyperledger composer-cli component.

The script *initNetwork.sh* needs to:

1. create package.json if it does not exist,
2. pack all sources into a bna-file,
3. create a network card for the *ChannelAdmin* as well as the *PeerAdmin* roles,
4. import that card,
5. install the new network,
6. start the new network,
7. and finally import the network admin card generated by the starting process.

[29] https://hyperledger.github.io/composer/v0.19/tutorials/acl-trading

Basically, the script will provide a running business network together with composer cards which are connection profiles used to access the network.

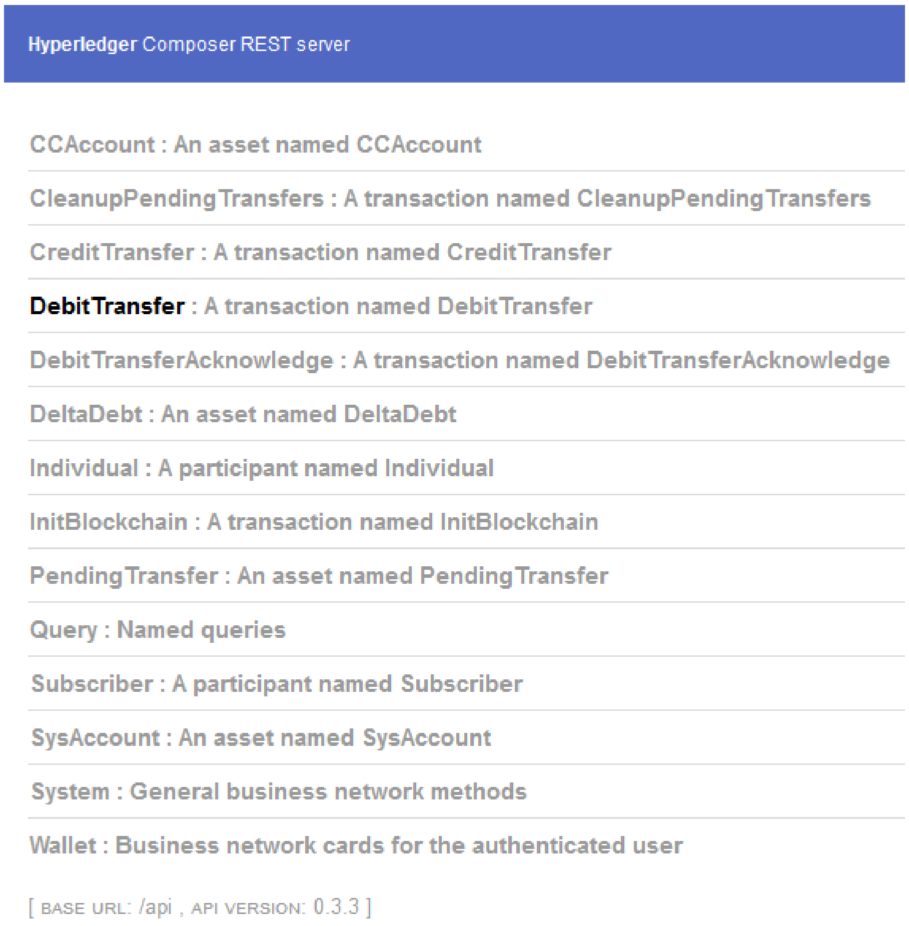
During development the network usually needs to be deployed a couple of times to check and test the implementations. *updateNetwork.sh* was created for this purpose. It reduces these 7 steps necessary for initial install to just 4 that handle an intermediate update. These steps are:

1. increase the version of the business network implementation,
2. re-pack the changed sources into a .bna file with a higher version tag,
3. install the new .bna file on the network,
4. and at last upgrade the network to the new version.

3.5 REST Server

A REST server used for connections of client applications can be provided by the Hyperledger Composer REST Server CLI-Tool.[30] The REST server is able to connect to the business network and provides access to all assets, transactions, and queries available on the network.

For INTERLACE all the configurations necessary to run this server are gathered inside *startRestServer.sh*. The exposed REST API is shown in Figure 3.8 and is built with Swagger tools.[31]



Hyperledger Composer REST server

CCAccount : An asset named CCAccount

CleanupPendingTransfers : A transaction named CleanupPendingTransfers

CreditTransfer : A transaction named CreditTransfer

DebitTransfer : A transaction named DebitTransfer

DebitTransferAcknowledge : A transaction named DebitTransferAcknowledge

DeltaDebt : An asset named DeltaDebt

Individual : A participant named Individual

InitBlockchain : A transaction named InitBlockchain

PendingTransfer : An asset named PendingTransfer

Query : Named queries

Subscriber : A participant named Subscriber

SysAccount : An asset named SysAccount

System : General business network methods

Wallet : Business network cards for the authenticated user

[BASE URL: /api , API VERSION: 0.3.3]

Fig. 3.8: **REST GUI provided by composer-rest-server using Swagger**

[30] https://hyperledger.github.io/composer/latest/reference/rest-server
[31] https://swagger.io/

3.6 Web Application

Figure 3.9 shows an illustration of a web page generated for the INTERLACE business network implementation. It was created using generator-hyperledger-composer,[32] which is a Yeoman[33] module.

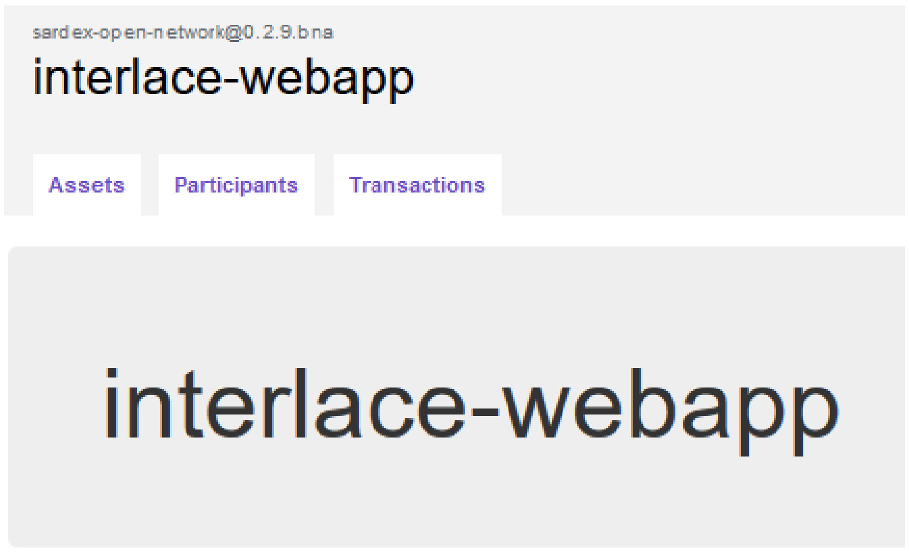
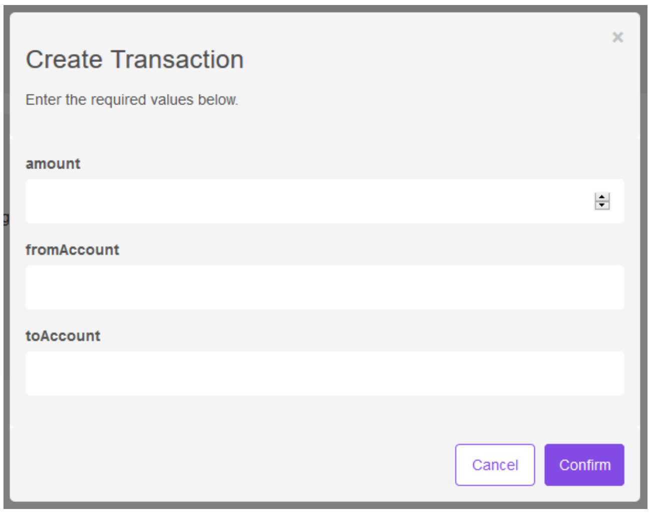


Fig. 3.9: **An AngularJS-based web application**

In this web application it is possible to list, add, and edit assets and participants, as well as submit transactions to the network. It is implemented in AngularJS[34] and connects to the REST server which is, as mentioned, also provided by the Composer Tools suite. Consequently, it is possible to manage assets like *SysAccount*, *CCAccount*, *PendingTransfer*, and *DeltaDebt*. Participants like *Subscriber* and *Individual* are using these assets, whereas transaction *CreditTransfer*, *DebitTransfer*, *DebitTransferAcknowledge*, and *CleanupPendingTransfer* handle how the assets are processed during application access.


Create Transaction

Enter the required values below.

amount

fromAccount

toAccount

Fig. 3.10: **Credit/Debit Transfers**

[32] https://www.npmjs.com/package/generator-hyperledger-composer
[33] http://yeoman.io/
[34] https://angular.io/

With the appropriate permissions, like we have with the admin user, assets may be changed directly, without calling any of the mentioned transactions which normally take care about updating the chain consistently. Further, regardless of how the assets are changed (i.e. using custom transactions or just executing changes directly), all manipulations to the chain are recorded to the chain and clearly traceable.

Figure 3.10 shows the graphical interface for an invocation of a credit/debit transfer. It picks up a *fromAccount*, *toAccount* and the amount that needs to be transferred. As explained in Section 3.4, for a *DebitTransfer* to be performed the owner of the *fromAccount* needs to send a *DebitTransferAcknowledge* confirmation.

Chapter 4

Conclusion and Final Thoughts

Eduard Hirsch

This last chapter discusses the goals reached as well as the problems encountered during the development of the INTERLACE prototype. Additionally, it emphasises possible enhancements necessary and issues which need to be taken into account in order to bring the prototype to production level. Finally, it discusses parts that could not be finished as well as after-INTERLACE goals.

4.1 Best Practices, Falsey Values and Pitfalls

When using Hyperledger Composer but also when connecting to Hyperledger Fabric directly, it is important to pay attention to some points. Developers have to be aware that working with chaincode or smart contracts is quite different from accessing data as usual in a RDBMS,[35] even though up-to-date frameworks shield a lot of the complexity underneath. Thus, even though access to the data structures and virtual machine/execution capabilities looks similar from a developer's point of view, it is necessary to account for some peculiarities related to blockchain-based technologies, because of their highly distributed nature. In the following sections we discuss and attempt to clarify some of these peculiarities.

4.1.1 Deterministic Execution

One of the most important things to keep in mind when writing chaincode applications is the deterministic execution of transactions. Although it seems quite obvious at first, it can be quite challenging to achieve.

One example is the **generation of IDs**: in a standard database environment simple locking mechanisms are in place to ensure correct primary keys for entries in a table. For blockchains which live in a distributed, consensus-based system it is problematic to create identifiers over chaincode execution. The reason is that each peer processing a transaction would compute a new ID completely independently, and most likely at the about the same time. Thus, if such a key-/ID-generation created different IDs on different clients, it might not be possible to reach consensus; thus, although nothing is actually wrong with the transaction itself, the resulting blockchain states in each peer would be different and therefore the last action would be rolled back. This would be especially hard to deal with during race conditions, and even more difficult to find out why a particular problem has occurred.

One solution to this problem could be to generate the ID from the client and pass it to the transaction as parameter. This would result in a much safer creation process which, additionally, is much faster during execution.

Another problem is posed by the use of **random numbers**. Since such calculations would reach different results on the various peers of the network, creating random numbers in chaincode executions would cause sever problems when creating IDs from those numbers or randomizing

[35] Relational Database Management System

decisions based on them, causing the results to be non-deterministic. Further, since it is not possible to know when a peer in the network will receive a new transaction, also the **creation of a date** or a timestamp contains an intrinsically random factor. Thus, dates created by different peer nodes during chaincode execution are most likely at least a bit different from each other, and when written into an e.g. asset can cause fork-inducing blocks, i.e. blocks with different hashes on the various nodes that would therefore need to be rolled back when detected.

Listing 4.1 shows an example of a chaincode function which creates a new date in line 8. The variable $currentDate$ will be filled with the current timestamp of each node's operating system.

```
 1  /**
 2   * CreditTransfer transaction
 3   * @param {net.sardex.interlace.CreditTransfer} transfer
 4   * @transaction
 5   */
 6  async function CreditTransfer(transfer) {
 7    [...]
 8    let currentDate = new Date(); //incorrect!
 9    [...]
10  }
```

Listing 4.1: **An example of chaincode with wrong determination of the current date**

This implementation will lead to inconsistent values of different peer nodes, which may result in various problems. For example, a possible scenario is that the date has been truncated to contain only day, month and year (no hour, seconds, ...), such that execution of the same transaction on different peers is likely to happen on the same day. In such a case, peers are able to reach consensus most of the time. However, validation of a transaction on different blockchain nodes would not be possible if it happens on different days (if e.g. execution is delayed or at the end of a day) or if it is (re-)checked retroactively.

To explain further, chaincode running on peer A may produce a date of 12 April 2018 at 23:59 in the evening. If peer node B receives the same transaction at 00:00 the next day, it will produce the date 13 April 2018. Also, if e.g. somebody wants to check if the blockchain is in a consistent state, a transaction might be re-executed at any point in time after 12 April 2018 (staying with this example). Then the chaincode needs to run again and a date already written into the chain will be definitely different from the date produced during the re-execution. Consequently, those examples will lead to different outcomes on different peer nodes and therefore will trigger a rollback of the transaction or mark the whole chain as invalid from the point in time where that particular transaction had been appended to the ledger.[36]

The solution to these kinds of problems is to provide the possibly changing factor as an input parameter instead of creating it inside of the chaincode. Input parameters stay the same once a transaction proposal has been accepted and written into the chain. Thus, for every re-execution or validation they may be picked from there, allowing a re-run of the code that results in the same outcome every time when called. Therefore the re-generation of the the block hash will give the exact same hash as the "previous Hash" attribute of the next block.

Given that in our implementation we are using Hyperledger Composer, the solution to the date problem could be to add an additional attribute for the abstract transaction $Transfer$ inside of our CTO-model definition, like adding $DateTime\ timeInitiated$. Then it would be necessary to pass a

[36] Ledger structure: https://hyperledger-fabric.readthedocs.io/en/release-1.3/ledger/ledger.html

date value to the transaction for the *timeInitiated* property when invoked by a client which then could be read from the transaction data during chaincode processing.

However, in the case of the date Hyperledger offers a better solution. In the sequence diagram of the transaction flow[37] that is prescribed by the Hyperledger protocol, the very first step when proposing a transaction is to submit a *PROPOSE* message to the endorsing peers. The propose message contains, among other things, the current timestamp of the transaction that, if validated, is eventually stored into a block. Thus, our prototypical INTERLACE implementation enables this timestamp to achieve the same goal as the propagation of the current date using the Transfer CTO-type. The code example in Listing 4.2, finally, shows how the date may be read from the transaction data.

```
/**
 * CreditTransfer transaction
 * @param {net.sardex.interlace.CreditTransfer} transfer
 * @transaction
 */
async function CreditTransfer(transfer) {
  [...]
  let currentDate = transfer.timestamp //correct!
  [...]
}
```

Listing 4.2: **An example of chaincode with correct determination of the current date**

Final note: Chaincode needs to be executed deterministically and has to reach, given its input parameters, the same result(s) on all the peer nodes at any point in time. Consequently, many parameters cannot be generated by the chaincode directly but need to be provided as parameters to a transaction. But since this means that the parameters are creatable on the client-side only, it is also necessary to implement the corresponding logic in a way that prevents them from being used to fool the system or bring it into an inconsistent state.

4.1.2 Network Upgrade

Traditionally, updating a database after changes to the database model may be quite cumbersome due to the presence of new fields, foreign keys, and many other similar issues. However, various methods and strategies are in place for handling these issues. On the other hand, since blockchains are usually spread over various peers that store replicated data, in contrast to traditional databases it is inherently more difficult to change the distributed structures to fit some new schema necessary to add a new feature or functionality.

Updating or changing values of asset attributes, as well as getting executable code and data structures ready for the next version of the network, may run into several problems. In particular, in order to maintain consistency and reliably achieve all fixes it is necessary to apply all changes to the network in such a way that upgrading the business network (CTO-File, chaincode, ...) and fixing the asset values happen in the right order, and while nobody else is able to interfere. This can be achieved by executing the upgrade as a single atomic transaction or, if that's not possible, by blocking common user access during deployment.

Thus, it is quite important to get everything right from the beginning, as it can be extremely difficult to apply certain changes. Consequently, it is highly advisable to think about possible

[37] https://hyperledger-fabric.readthedocs.io/en/release-1.3/arch-deep-dive.html#swimlane

changes and future scenarios together with possible side-effects and costs before first deployment to production systems.

4.1.3 Hyperledger Composer Specifics

Composer can be seen as a rapid-development approach. Its API sits on top of the Hyperledger Fabric framework. It tries to shield complexity from the user and aims to significantly reduce the time necessary to create distributed applications for Hyperledger Fabric.

For the development of the INTERLACE transactional service this meant that:

- models could be transferred easily,
- chaincode could be based on these models,
- a REST Server was supplied and
- a web-application generator was available.

Consequently, it was an ideal framework for prototyping. In addition, once a suitable cloud provider has been found, spinning up a network is straightforward. When hosting a network without an external provider who knows how to run these networks, much of the initial speed in creating it might be lost because, in such a case, a much deeper understanding of the whole architecture is necessary. Thus, also a greater effort is needed to get a ready implementation to work.

Note: For production systems it would be necessary to change to a plain Fabric implementation because Composer is still at quite an early stage of development, and there are even rumours that it may be discontinued. The reason is that some of the new features of the Fabric releases are deviating from the structures implemented by Composer. So it is becoming increasingly difficult for Hyperledger Composer developers to provide new features and functionalities available for Fabric.

4.2 Identity Management

As mentioned in previous sections, we skipped Identity Management for the prototype in order to simplify the environment, decrease development efforts and therefore offer a relatively easier access to an example mutual credit system, hoping that this would make it easier to understand the basics. Nevertheless, it would be possible and quick to create additional participants and grant them access to the network in addition to just the currently used admin account. For example, an identity card can be issued for the first participant defined in transaction "InitBlockchain" which is identified by "m1". The example below illustrates how this might be done:

```
composer identity issue -c admin@tutorial-network -f m1.card -u m1 -a "resource
  ↪ :net.sardex.interlace.Individual#m1" -x true
```

This identity card is facilitated by the REST server and corresponds to a participant in the business network. But, when accessing the REST server, before a user can act with that identity it is necessary for him/her to log in first and authenticate his/her identity in some sort of way. To do so composer-rest-server uses an authentication middleware implemented in JavaScript called *Passport.js*.[38]

The passport middleware offers many different authentication strategies and is quite a mature Open Source framework. An example of how to authenticate with OAuth and GitHub can be found

[38] http://www.passportjs.org/

on the Composer documentation website;[39] but also different schemes can be used, like the JSON Web Tokens (JWT[40]) which is mentioned in the issue tracker of the Hyperledger Composer GitHub page.[41]

Passport is a commonly known middleware for authentication and may be applied also to different scenarios. Thus, it is usable well also if Composer is swapped for another technology.

4.3 Future Scenarios

Because we are using Hyperledger Fabric, extending the network is only a matter of changing the configuration. The approach taken for the INTERLACE prototype, shown in Figure 4.1, creates a new organisation for every payment circuit that deals with a new specific region. Consequently, each region would be responsible for hosting their own peers and ideally also their own certification authority (CA). However, certificates might be still handled by the Sardex CA.

Sardex, in this scenario, is providing the network architecture and transferring the know-how about how to run a circuit, which would result in a model similar to a franchise. Although the clients and various other visible graphical interfaces may be branded in various ways, the underlying platform infrastructure would be predefined by Sardex in order to make the network work consistently.

The orderer will be handled by Sardex as well, but later might be handed over to an e.g. non-profit organisation representing the circuit and enforcing fair and clearly defined rules within a governance framework defined in collaboration with the circuit members themselves. To ensure a reliable system and a high performance throughput, an orderer could be clustered locally but also over various regionally separated areas.

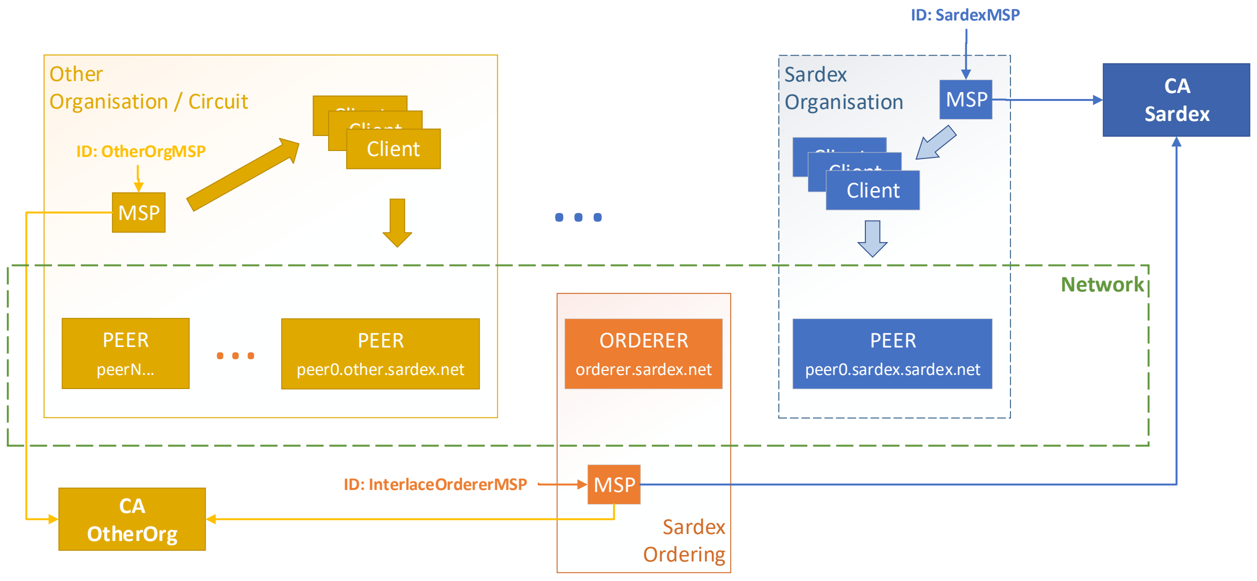


Fig. 4.1: **Extended Network Structure**

[39] https://hyperledger.github.io/composer/latest/integrating/enabling-rest-authentication
[40] https://jwt.io/
[41] https://github.com/hyperledger/composer/issues/2038

4.4 Final Review and Open Points

Our DLT application prototype forms a stable and scalable basis for a reliable payment circuit. In fact, although the prototype was developed with Hyperledger Composer, which IBM may discontinue support for, any subsequent implementation in Hyperledger Fabric only will now be much easier to realize. This is also because newer versions of Fabric support a node.js SDK,[42] which would even allow the transfer of JavaScript code bits.

In summary, the goal of creating a reliable DLT has been achieved, and forms a necessary foundation for the various other services provided in future for the production-level business solutions created by and with Sardex.

4.4.1 Open Points

The **GDPR**[43] directive that came into effect in the first half of 2018 represents a challenge for blockchain solutions and, therefore, also for the INTERLACE project. It is necessary that personal information is not exposed to other parties unless necessary and unless it is done in agreement with its owners. In addition, each piece of information collected for the user or provided by the user needs to be deletable if the user requests it, and it must certainly be possible to look up upon request exactly what data was collected.

Information is reliably stored inside of a blockchain. Thus, looking up information is actually not an issue. Rather, one of the challenges is to keep it secret from being read by other parties, since standard blockchain approaches copy everything to every peer. Another challenge is the immutability of most blockchains, since GDPR enforces the right to be forgotten. The privacy aspect is solved currently by making the actual chain only accessible by the peers that are owned by the organisation running the local circuit. In this configuration, the clients that perform transfers have restricted access and cannot see the whole blockchain.

In later stages it may be possible for every business/party participating in the payment network to access the blockchain directly, i.e. to run a node. In this case the so-called *SideDB*[44] can be taken into consideration because it is a way offered by Hyperledger Fabric to store information which is only known by the respective client and only shared if permitted by the client.

SideDB also solves the problem of GDPR-relevant data because, first, it is only exposed "to whom it may concern" and, second, it can be purged without making the chain invalid. The reason is that in SideDB only hashes of transactions and data are stored on the chain, which still makes it verifiable by regenerating the hash and comparing every time a datum needs to be verified. Normally the data only needs to be provided if specifically asked for, e.g. in case of an (external) audit.

The second point which needs further effort is the part of the testing coverage where the **ASIM implementations** are **tested** against the actual implementation of the prototype and later against the production-level implementation. These tests will be covered in deliverables D4.1 and D4.2 whose future versions will be completed after the end of the project.

[42] Software Development Kit
[43] General Data Protection Regulation [11]
[44] https://hyperledger-fabric.readthedocs.io/en/latest/private-data/private-data.html

References

References

1. E Androulaki, A Barger, V Bortnikov, C Cachin, K Christidis, A De Caro, D Enyeart, C Ferris, G Laventman, Y Manevich, S Muralidharan, C Murthy, B Nguyen, M Sethi, G Singh, K Smith, A Sorniotti, C Stathakopoulou, M Vukolic, S W Cocco, and J Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *EuroSys ï£¡18: Thirteenth EuroSys Conference, April 23-26, 2018, Porto, Portugal*, New York, NY, USA, 2018. ACM. URL: `https://doi.org/10.1145/3190508.3190538`.
2. E Börger and R Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. New York: Springer-Verlag, 2003.
3. P Dini, E Börger, E Hirsch, T Heistracher, M Cireddu, L Carboni, and G Littera. *D2.1: Requirements and Architecture Definition*. INTERLACE Deliverable, European Commission, 2017. URL: `https://www.interlaceproject.eu/`.
4. P Dini, G Littera, L Carboni, and E Hirsch. *D2.2: Iterative Architecture Refinement*. INTERLACE Deliverable, European Commission, 2018. URL: `https://www.interlaceproject.eu/`.
5. P Dini, G Littera, L Carboni, and E Hirsch. *D2.3: Final Architecture*. INTERLACE Deliverable, European Commission, 2018. URL: `https://www.interlaceproject.eu/`.
6. Thomas Erl. *Next Generation SOA: A Real-World Guide to Modern Service-Oriented Computing*. Prentice Hall, 2014.
7. Kevin Forsberg and Harold Mooz. The relationship of system engineering to the project cycle. In *INCOSE International Symposium*, volume 1, pages 57–65. Wiley Online Library, 1991.
8. E Hirsch, T Heistracher, P Dini, E Börger, L Carboni, M L Mulas, and G Littera. *D3.1: First Demonstrator Implementation*. INTERLACE Deliverable, European Commission, 2018. URL: `https://www.interlaceproject.eu/`.
9. Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.
10. Anthony O'Dowd, Venkatraman Ramakrishna, Petr Novotny, Nitin Gaur, Luc Desrosiers, and Salman Baset. *Hands-On Blockchain with Hyperledger*. Packt Publishing, 2018.
11. European Union Parliament and Council. GDPR - general data protection regulation. `https://gdpr-info.eu`, 2018. [Accessed: 2018-11-27].