POLITECNICO DI TORINO

MASTER THESIS

Blockchain for Education
Case Study on Hyperledger Fabric

Author:
Sara GIAMMUSSO

Academic Supervisor(s):
Prof. Paolo GARZA
Prof. Elia PETROS

Internship Supervisor:
Carlos Ernesto CANALES

A thesis submitted in fulfillment of the requirements
for the Master of Science MSc in Computer Engineering

in the

Control and Computer Engineering Department

16th April 2019

To my mother Silvana, my father Totò and my brothers Davide and Simone, for their love and endless support and for always being there where I need it.

"Every person that you meet knows something you don't; learn from them."

H. Jackson Brown Jr.

Abstract

Evoke is an award-winning online educational experience, which uses storytelling and game mechanics to prepare young people to become social innovators who create solutions that address global "grand challenges" (e.g. poverty, hunger, water scarcity). Evoke enables students to demonstrate their learning by submitting "evidence" of their activities and provides incentives, rewarding students through tokens that can be used in the platform marketplace.

The goal of this thesis is to design a prototype to test the applicability of blockchain within this use case, designing a solution that can make the project scale across different countries, and trace the distribution of donor funds.

A demo video of the PoC can be found here.

Sommario

Evoke è una piattaforma didattica online pluripremiata, che utilizza la narrazione e le meccaniche di gioco per preparare i giovani a diventare innovatori sociali capaci di creare soluzioni che risolvono le "grandi sfide" globali (ad esempio povertà, fame nel mondo, crisi idriche). Evoke consente agli studenti di dimostrare il loro apprendimento inviando "prove" delle loro attività e fornisce incentivi economici, ricompensando gli studenti attraverso token che possono essere utilizzati nel market place della piattaforma.

L'obiettivo di questa tesi è di progettare un prototipo per testare l'applicabilità delle blockchain in questo caso d'uso, definendo una soluzione che possa permettere al progetto di scalare in diversi paesi e allo stesso tempo tracciare la distribuzione dei fondi dei donatori.

Un video demo del prototipo puo' essere trovato qui.

Résumé

Evoke est une expérience éducative en ligne primée qui utilise des mécanismes de narration et de jeu pour préparer les jeunes à devenir des innovateurs sociaux qui créent des solutions qui répondent aux "grands défis" mondiaux (par exemple, pauvreté, faim, pénurie d'eau). Evoke permet aux étudiants de démontrer leur savoir en soumettant des "preuve" de leurs activités et fournit des encouragements, récompensant les étudiants par des jetons qui peuvent être utilisés sur le market place de la plate-forme.

Cette thèse a pour objectif de concevoir un prototype permettant de tester l'applicabilité de la blockchain dans ce cas d'utilisation, de définir une solution qui permet au projet de grandir en différents pays et au même temps de suivre la répartition des fonds des donateurs.

Une vidéo de démonstration du prototype peut être trouvée ici.

Acknowledgements

I have to thank first of all the World Bank Group ITS Technology and Innovation team who introduced me to the world of blockchain and gave me the opportunity to work on this interesting topic. I am very grateful to Carlos Ernesto Canales and Maria Vargas for their help on the requirements definition and for their very important moral support when I needed it.
I also want to thank Paola D'Alessandro who has been the closest thing to a mom in the United States.

A special thanks goes to my professors and supervisors Paolo Garza from Politecnico di Torino and Elia Petros from EURECOM Campus of Télécom ParisTech which gave me several advises about this work.

Thanks to Collegio Einaudi, to have made Turin a second home, and particularly my Turin family: Giampaolo, Luca and Marco, I would never have obtained this results without their help and support. Moreover, I have to thank my girlfriends Alessia, Claudia, Giuliana, Roberta and Simona, for 10 years of standing by my side and giving me reasons to cheer.

Finally, last but not least I have to thank my mother Silvana and my father Salvatore, whose support, both moral and economical has made my education career possible, and to my brothers Davide and Simone who have always pushed me to do my best in everything.

Contents

List of Figures

List of Abbreviations

PoC	**P**roof **o**f **C**oncept
B4E	**B**lockchain **4**for **E**ducation
DLT	**D**istributed **L**edger **T**echnology
WBG	**W**orld **B**ank **G**roup
ITSTI	**I**nformation **T**echnology **S**ervices **T**echnology **I**nnovation
PoW	**P**roof **o**f **W**ork **PoS**
Proof **o**f **S**take **PKI**	**P**ublic **K**ey **I**nfrastructure
A	Client **A**pplication
S	**S**mart **C**ontract
CA	**C**ertificate **A**uthority
P	**P**eer
L	**L**edger
O	**O**rderer
CP	**C**hannel **P**olicy
NP	**N**etwork **P**olicy
CRL	**C**ertificate **R**evocation **L**ist

Chapter 1

Introduction

1.1 Evoke: an overview

The world of work is changing dramatically, and how education is designed and delivered needs to change just as dramatically. Nearly 85% of the world's young people live in lower-income countries and fragile states. The probability to be unemployed for young people is 4 times higher than for adults.

To engage young people where they spend their time - reading graphic novels, engaged in social networks, immersed in game environments, the World Bank created and deployed Evoke, an award-winning social networking game that empowers young people to think about and collaborate with their peers locally and globally to solve the world's most urgent social problems (e.g., displacement, hunger, poverty,water scarcity)[2]. (See the original Evoke trailer at this link)

The Evoke project is designed to help young people developing an understanding of these complex challenges, acquire 21st century skills (e.g., creativity, collaboration, critical reflection), socio-emotional skills (e.g., curiosity, empathy, generosity), and gain the confidence to experiment, collaborate, and create innovative solutions.

1.1.1 System design

The primary target audience of the game is young people, defined as ages 17-25. This spans the upper secondary level and university populations, as well as out-of-school and working youth.

Evoke involves 5 different participants:

- Donors - they add funds to the overall Evoke trust fund;
- Vendors - they participate in the Evoke marketplace by accepting tokens in exchange for their goods and/or services;
- Students - they engage with Evoke missions submitting evidences of their work thus earning tokens they can spend in the Evoke marketplace;
- Mentors - they engage with students reviewing, commenting and evaluating their evidences, thus earning tokens they can spend in the Evoke marketplace;
- Campaigns - they create new content to support learning objectives.

1.1.2 The challenges

The original iteration of EVOKE, URGENT EVOKE, launched at the TED conference in early 2010 by co-creator and noted game designer Jane McGonigal, generating much media interest, it ran as a 10-week "crash course in saving the world"

from February through May. Geared for students across Africa, this first version of EVOKE was in English. It won a top award at the 8th Annual Games For Change Festival [5].

Evoke had other four iteration but it encountered different challenges.
Currently, Evoke is a standalone, in-country effort led by a not for profit agency in partnership with an academic institution. Over a period of 12 weeks, students play Evoke in a team to complete their class project. They receive new missions each week on the website, and submit a blog post as evidence of completion to receive their incentive. These incentives can be bus passes or telephone minutes.

Every time a game is to be launched in a country, the complete system set-up is repeated creating an obstacle and persistent challenges.
Additionally, tracing the impact and distribution of donor funds is a global challenge. The time taken from when donor funds are received when the donation reports can be sent back to the donors can range anywhere between a few months to a year.

The WBG ITSTI Team has been engaged to re-design Evoke using emerging technologies, developing a prototype focused on two key Evoke concepts that could be potentially transformed by deploying a blockchain platform:

1. *Transparent funding* - Donors can decide to allocate funds to a given campaign. After the donation they must be able to observe how donations have been spent. Traceability could be reported in real-time.

2. *Token economy* - Students and mentors complete pre-assigned tasks. On providing evidence of the completion of the tasks, these agents are allowed to collect their incentives, i.e., coins that can be used in the game marketplace. These coins cannot be exchanges for fiat currency in the real world.

1.2 Blockchain for education: Related works

Due to its ability to increase transparency and trust blockchain has already been largely used in the education field. Since transactions are secured and linked to each other in a chain of blocks, data are very hard to tamper or erased. Therefore, blockchain have been largely used in validating identity management, specifically to verify that we are who we claim to be.

In a always more competitive employment world, always more qualifications are needed and a lot of people started lying about the certification owned. However, thanks to the ability of blockchain to provide a tool to verify people's education background and validate candidate's CV, universities such as MIT already started to issue diplomas in a digital format together with the paper-based copy.

New apps, such as *Blockcerts Wallet*, provide to students the possibility to share a verifiable and tamper-proof digital version of their diplomas with employers or other institutions.

Another example is *Gradbase*, which has already been used from University College London to issue degree certificates for some courses. Students are provided with a QR code that they can share directly on LinkedIn. Employers can then easily verify the educational credentials through the app.

Sony Global Education instead, is bulding a blockchain network based on Hyperledger Fabric to create an online depository of education records. The system put

together data from different universities, thus creating a one stop education profile.[4]

Opet Foundation's product is an education companion chatbot app that aims to provide users with lessons to help them prepare for examinations. Apart from answering student queries, the app is designed to recommend worksheets and problems for users to solve. It will then create education profiles for users, tracking their learning speed and grades. This information will be placed on the blockchain, allowing institutions to evaluate students based on their progress.

Opet Foundation can help bridge the class divide that limits opportunities for students. Indeed, everyone takes the same national or matriculation examination, but people who are underprivileged or don't have the access to the right kind of finances, end up falling behind or having to struggle a lot in admissions.

For example, a student who is adept at advanced mathematics could be expected to fare well in engineering. Opet's app can recommend students with related aptitudes to academic institutions that offer courses that they might excel in. Finally, Opet can recommend students to universities based on their education profile. [10]

1.3 This Master Thesis

This work has been carried out during a 6-months university internship at the World Bank Group ITS Technology and Innovation Team (Washington D.C., US) under the supervision of Carlos E. Canales (WBG), professor Elia Petros (EURECOM Campus of Téléecom ParisTech) and professor Paolo Garza (Polytechnic of Turin).

Aim of this Master Thesis is to study how can Blockchain impact transparency and scalability of Evoke, a World Bank project aimed to inspire youth around the world to develop a passionate curiosity for learning and to produce a Proof of Concept that shows how Evoke architecture can be re-designed using blockchains.
The chosen blockchain framework, Hyperledger Fabric, is an open source enterprise-grade permissioned distributed ledger technology (DLT) platform, established under the Linux Foundation.

During this period of Master Thesis, the author has deeply explored from a theoretic point of view the Hyperledger Fabric architecture, then she has designed the PoC back-end using Hyperledger Composer, a set of JavaScript-based tools that simplify and expedite the creation of Hyperledger Fabric blockchain applications, as well as the PoC front-end, implementing an Angular application.

A link to the demo can be found here.

1.3.1 Research Questions

This Master Thesis addressed two key research questions:

> *Research Question 1*: How can B4E ensure donated funds are properly maintained and manages (i.e. 100% of designated funds flow to educational purposes)?
>
> *Research Question 2*: How can B4E scale in as many countries as possible minimizing the duplication effort?

1.3.2 Outline

This Master Thesis is organized as follows:

Chapter 1 The introduction to the Master Thesis, providing a brief overview on the WBG Evoke project, including an excursion into the current situation and challenges; an overview on the main blockchain frameworks; an overview on the main requirements of the PoC, which is the topic this Thesis is focun on; together with the outline of the Master Thesis itself;

Chapter 2 A deep description of the the basic theoretical knowledge about blockchain, Hyperledger Fabric and Hyperledger Composer;

Chapter 3 A description of the proposed solution, including the business network definition together with the chosen network architecture;

Chapter 4 The conclusion of the Thesis, presenting future works and final remarks.

Chapter 2

Theoretical Background

2.1 Blockchain Technology

Blockchain is a shared, replicated transaction system which is updated via smart contracts and kept consistently synchronized though a collaborative process called consensus.

Blockchains are a form of distributed ledger technology (DLT), digital ledgers shared across multiple locations. The main feature of distributed ledgers is the absence of a central administrator or a central data storage. Due to the lack of a trusted third party DLTs use consensus algorithms to ensure the correct replication of data across nodes.

The blockchain concept finds its origin in the paper "How to time-stamp a digital document" by Haber and Stornetta [6]. They did not necessarily coined the term "blockchain", but they aimed to certify when a document was created or changed in a secure and immutable way, impossible to forge even with the collusion of the time-stamping service.

However the blockchain concept became much more famous after the publication of the paper "Bitcoin: A Peer-To-Peer Electronic Cash System" [9] whose author is known by the pseudonym, Satoshi Nakamoto. Originally devised for the digital currency, Bitcoin, the tech community is now finding other potential uses for the technology.

In order to explain how a blockchain work let's use Bitcoin as example.

In a blockchain data comes in blocks. Blocks are linked together in a chain, thus making difficult to tamper data.
Suppose to have different blocks of transaction data, as in Fig. 2.1.

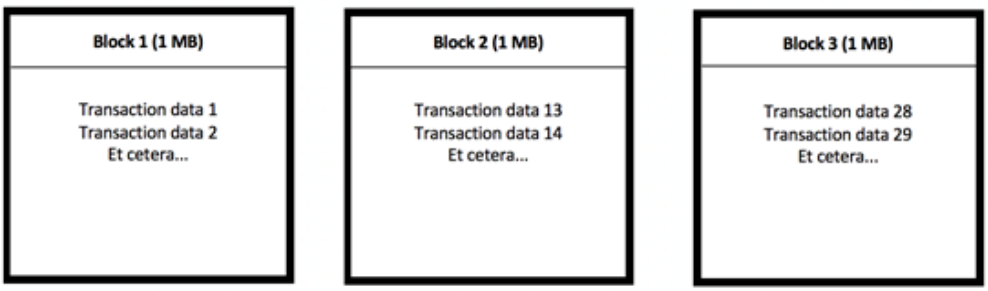


FIGURE 2.1: Chain of blocks

In order to link the blocks, every block gets a unique signature, generally obtained through a hash function, that corresponds to exactly the string of data in that

block. If anything inside the block changes, even just a single bit, the signature will be completely different.

Let's say that block 1 registers two transactions, transaction 1 (T1) and transaction 2 (T2), and let's say the signature for this block is "X32", as shown in Fig. 2.2.

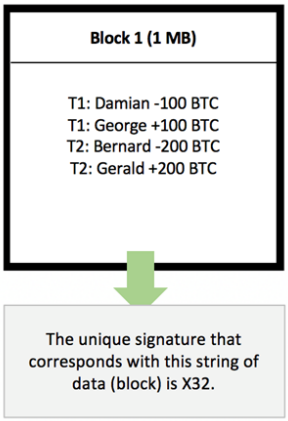


FIGURE 2.2: Single block example

Adding the block 1 signature to block 2, the two blocks are now not only linked together, but also in some way connected since the block 2 signature now depends also on the signature of block 1.

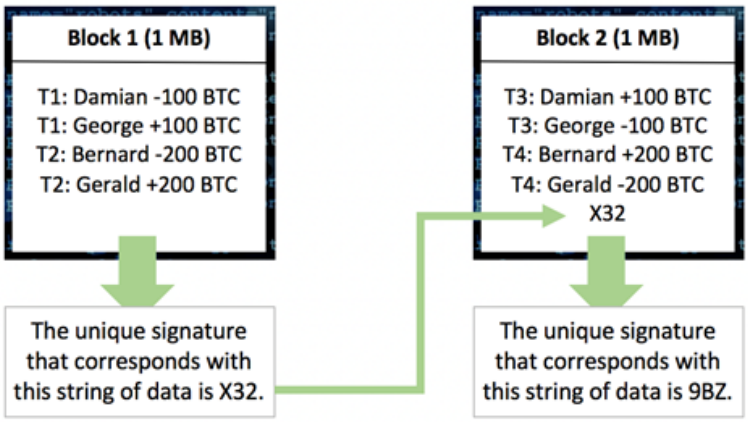


FIGURE 2.3: Block 1 and block 2 linked together

It is thanks to the signature that we can link blocks together. Let's now suppose to add block 3.

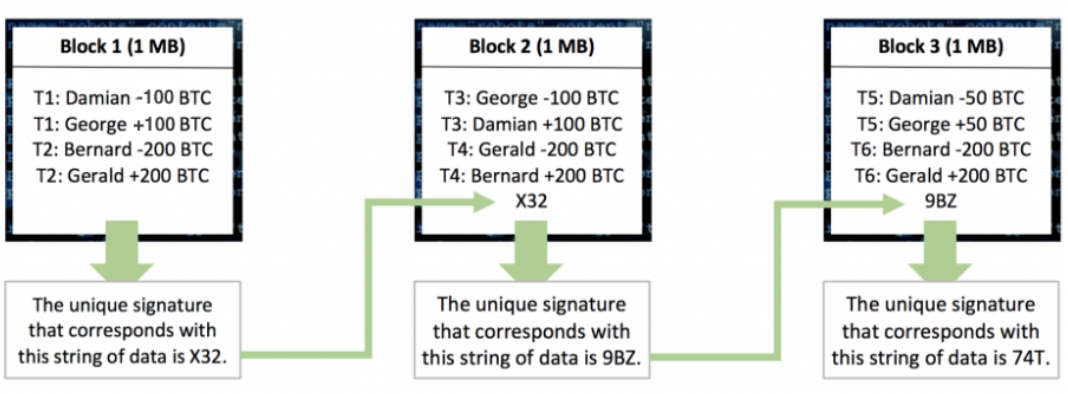


FIGURE 2.4: Block 1, block 2 and block 3 linked together

If we now tamper data in block 1, for example we alter the transaction between George and Damian and Damian now supposedly sent 500 Bitcoin to George instead of 100 Bitcoin, the string of data in block 1 will drastically change, and so will do block 1 signature, e.g. the signature will now change from "X32" to "W10".

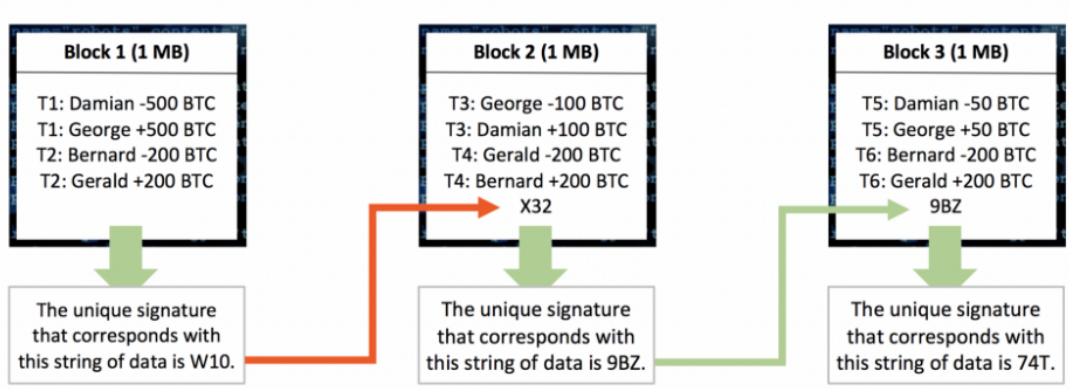


FIGURE 2.5: Block 1 has been altered, Damian supposedly sent 500 Bitcoin to George instead of 100 Bitcoin

The two blocks are no longer linked, and this allows the other node to avoid trusting that block. However, there is a chance for the attacker to stay undetected, if and only if the entire chain keep linked together, meaning that all the signatures from that block keep being linked. This is considered to be very hard to achieve, and the explanation really needs to understand how the signatures are created.

In blockhain, the signature is created by a *cryptographic hash function*. Hash functions are explained in more details in the following section. The process of adding a new block to the chain does not only involve the creation of the signature, but also the *consensus* protocol. Indeed, all the nodes participating to the blockchain must agree on the content and on the transaction order of a new block in the blockchain.

2.1.1 Hash functions

"A hash function is any function that can be used to map data of arbitrary size to data of a fixed size. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes" [7]. Mathematically a hash function is a function $h : X \to Y, |X| > |Y|$, where $|X|$ is the cardinality of the set X.

Cryptographic hash algorithms must fulfill at least 5 requirements:

1. Pre-image resistance: given a message digest y, it is computationally infeasible to find a message x such that $h(x) = y$;
2. Deterministic: given two messages x and x', $h(x) = h(x'), \forall x = x'$;
3. Ease to compute: for any message x, $h(x)$ is very quick to compute;
4. The avalanche effect: even a small change in a message x will reflect a huge change in the digest $h(x)$;
5. Collision resistant: it is computationally infeasible to find two messages x and x' such that $x' \neq x$ and $h(x') = h(x)$.

2.1.2 Consensus protocols

Consensus is the process of synchronizing the ledger transactions across the network's nodes to make sure that the different copies of the ledger are updated only when transactions are actually approved.

Transactions must be written to every copy of the ledger in the same order. This step requires the order of transactions to be established.

Consensus is a very researched area of computed science, and there are many ways to achieve it, each with different trade-offs. One example is the Practical Byzantine Fault Tolerance (PBFT) that provides a synchronization mechanism to keep each copy of the ledger consistent, even in the event of corruption. In Bitcoin, instead, transactions order is established through a process called mining where competing computers race to solve a cryptographic puzzle. The winner defines the order of the transactions in the ledger.

The biggest problem in reaching a consensus is known as "Byzantine Generals Problem".

The Byzantine Generals Problem Imagine that a group of Byzantine generals want to attack a city. They are facing three very distinct problems:

1. The generals and their armies are very far from each other thus making a centralized communication very difficult;
2. The messengers used by the generals are not reliable;
3. The city can be defeated if and only if the entire army is used to attack.

To coordinate the attack, a messenger is sent from the armies on the right of the castle to the one on the left of the castle, saying for example "ATTACK TUESDAY". However, it is possible that the armies on the left of the castle don't want to attack on Tuesday, because they prefer to do that on Wednesday, so they send a messenger saying "NO. ATTACK WEDNESDAY".

This is where we face a problem. Million of things can happen to that messenger, leading the army to an uncoordinated attack that will defeat.

Some consensus mechanism which can solve the Byzantine Generals problem are: Proof of Work (currently used in Bitcoin and Ethereum) and Proof of Stake (planned to be used in Ethereum in the future).

Proof of Work

Proof of work consensus protocol has been invented by Satoshi Nakamoto, Bitcoin's creator. In order to explain Proof of Work protocol let's now analyze the transaction flow in the case of Bitcoin blockchain.

Suppose to have a network of participants, called nodes. Nodes can be miners, as well as just people who want to transact on the network to send each other bitcoins or any other cryptocurrency. Fig. 2.6 describes a sample networks with 11 nodes: 4 miners and 7 people.

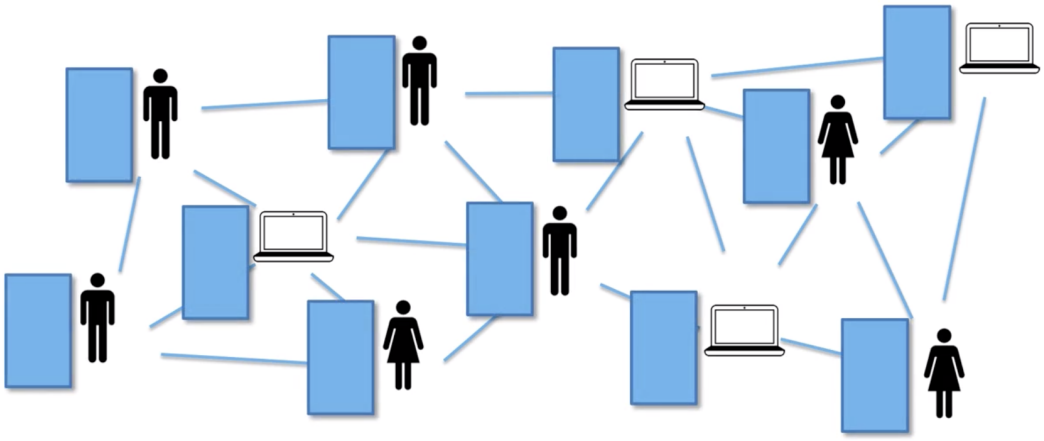


FIGURE 2.6: A sample networks with 11 participants

Every participant keeps a copy of the ledger, together with a local memory called *mempool* that acts as a staging area for transactions. In the case of Bitcoin blockchain blocks are added at a certain regularity, i.e. almost 10 minutes, but of course people transact with each other more frequently so the mempool is a staging area where transactions go before they are added to a block.

Let's now suppose that Susan wants to do a transaction.

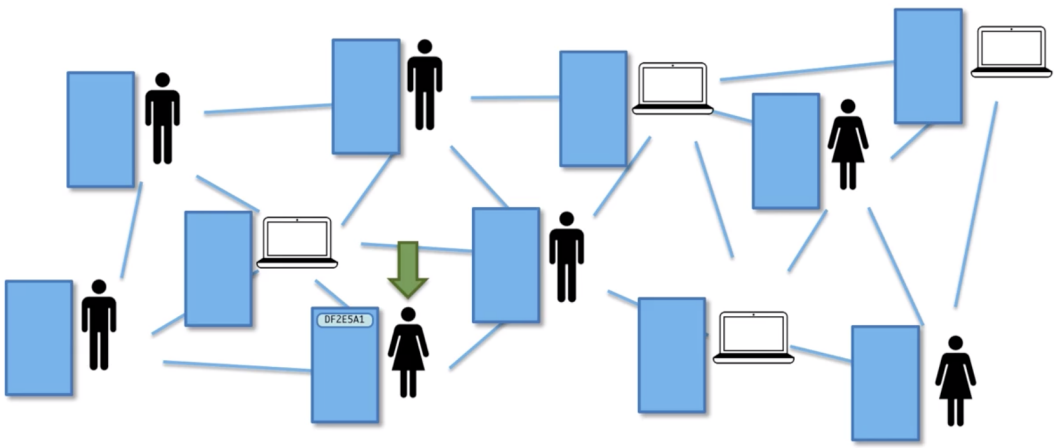
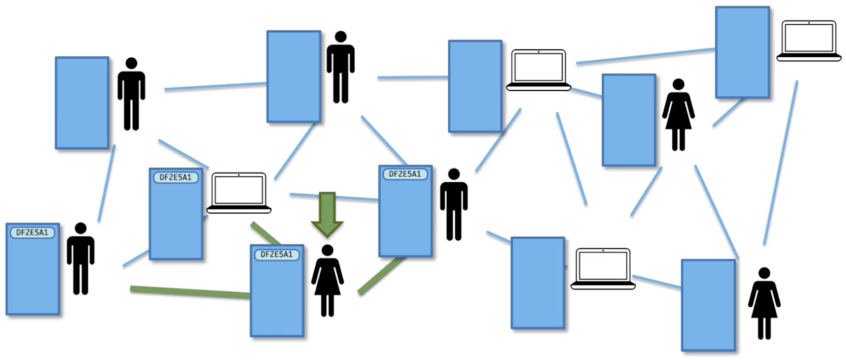


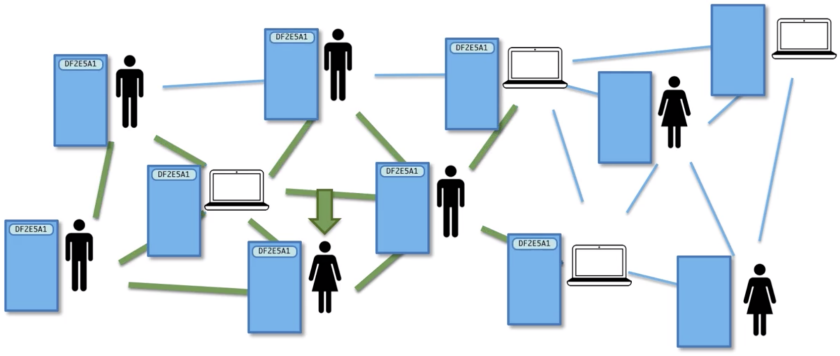
FIGURE 2.7: Susan does the first transaction. The transaction is added to her mempool

As explained in Fig. 2.7 the transaction is added to her mempool and then broadcasted across the network. She sends the transaction to her closest nodes, including

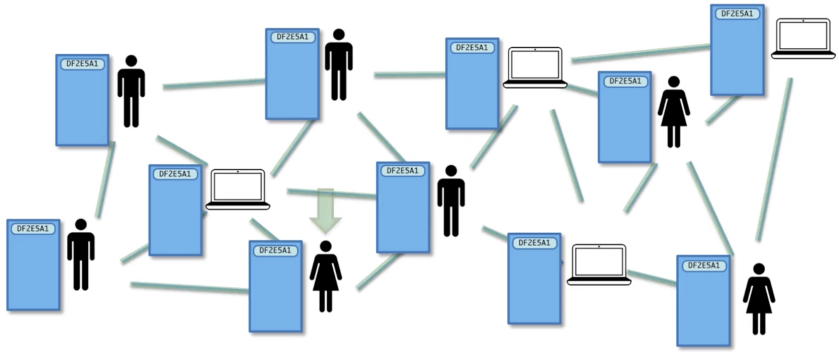
the miners. Every node conducts some checks to verify that the transaction is valid, if so, it adds the transaction to its mempool, and forward it to its neighbors so that the transaction gets added to every single mempool in the network. These steps are described in Fig. 2.8.



(A) Susan broadcasts the transaction to her closest nodes



(B) After validation, each node forwards the transaction to its neighbors



(C) The transaction is replicated in every mempool in the network

FIGURE 2.8: Bitcoin transaction flow: step 1

Now imagine that other participants (including the miners) do other transactions. As described in Fig. 2.9 the different mempool will contain more or less the same transactions but maybe in different orders.

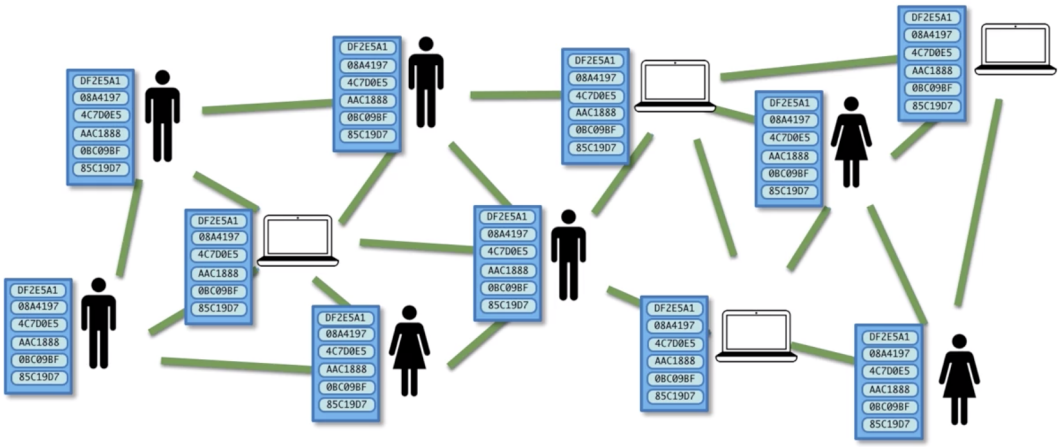


FIGURE 2.9: Other participants do more transactions

The miners will try to mine a new block once their mempool size reaches a certain threshold. The miner who succeed in creating a new block will be rewarded through some bitcoin. This is the reason why people are encouraged to participate in the Bitcoin blockchain. Now each miner has to create the block, adding some transactions as block data. In the case of Bitcoin, the criteria with which the miner chooses the transactions to add to a block is maximizing the bitcoin reward, that consists in the sum of the fees that the participant decided to pay for doing that transaction. This process is represented in Fig. 2.10.

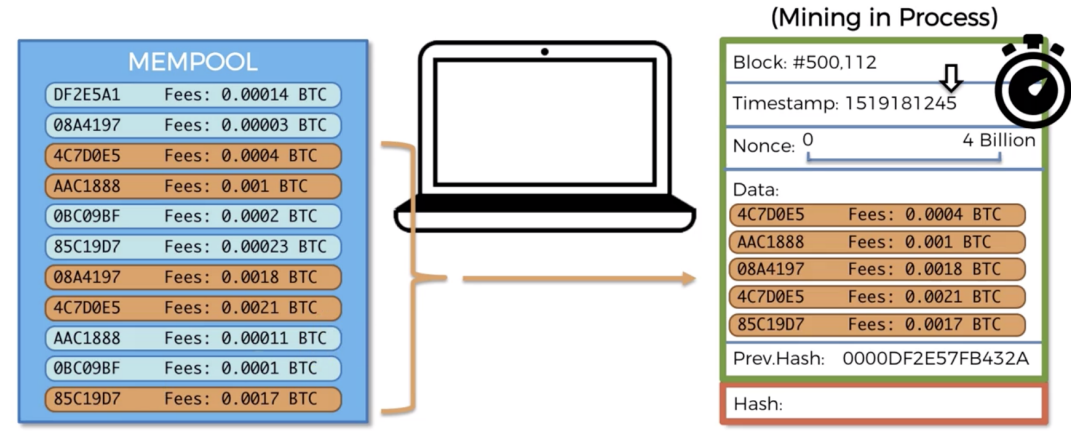


FIGURE 2.10: Bitcoin transaction flow: step 2

Now, miners must simply hash the block, however the hash must respect some rules, e.g. start with a certain number of zeros. Each block has a nonce field. The miner, after deciding which transactions to put in the block, selects a nonce, and tries to hash the block. If the hash does not respect the rule, the miner changes the nonce and tries again. The first miner who will find a hash starting with the right number of zeros will be the winner. This step is very computationally expensive and this is the reason why in the Bitcoin blockchain a block is mined more or less after 10 minutes.

Deciding the number of zeros allows to define the probability to have the right nonce. The higher the number of leading zeros, the smallest the probability that a miner will find the right hash, as explained in Fig. 2.11.

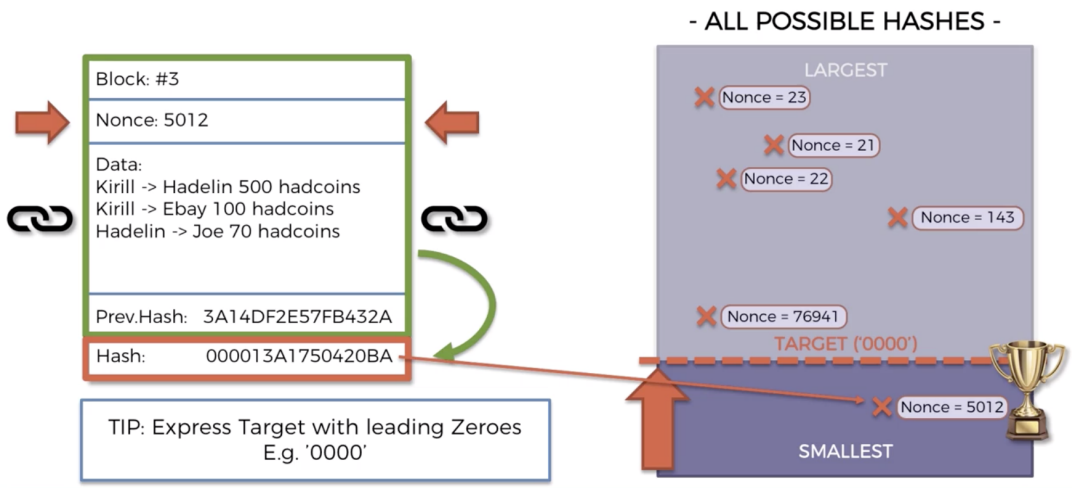


FIGURE 2.11: Bitcoin transaction flow: step 3

Once the miner finds the right nonce, it forwards the new block to its closest neighbors, and they do the same with their neighbors. Each participant, after validating the block, adds the block to his copy of the ledger. The validation is a really simple step, each participant must check that that block, with that nonce, gives a hash that respects the rule of the number of leading zeros.

Since hash function are never 100% collision free it is possible that two miners mine two blocks at almost the same time, so some parts of the network receive one block (or even more than one), and other parts another block. In this way, different part of the network may have different versions of the ledger. The PoW solves this conflict deciding that the longest chain wins and that the transactions of the rejected block go to the mempools again. That is the reason why, with the PoW, it is highly suggested to wait for at least other six blocks after the block containing the transaction, before considering that transaction valid.

Proof of Work solves the Byzantine Generals problem in the following way.

Suppose the army on the left send the message "ATTACK MONDAY" to the army on the right. Before sending the message they:

1. Append a random hexadecimal nonce to the original text;
2. Hash the text together with the nonce and the send the message only if the hash complies with some rules (e.g. it starts with 6 zero), otherwise they change the nonce and retry.

These two steps are very computationally expensive. Indeed, accepting only the hash starting with a given number of zeros allows to reduce or increase the probability that an hash will be accepted.

If something happens to the messenger and the message is tampered with, the hash will change too. If the generals on the right side see an hashed message that is not starting with the chosen amount of zeros they can simply cancel the attack.

It is possible that the message has been tampered with and the hash still starts with the right amount of zeros. To solve this issue the generals on the left send more than one message instead than one.
They append a random nonce to every message and hash it, and when all the messages' hashes are acceptable they append another nonce to the resulting hash and hash the cumulative message again.

If the messenger does get caught by the city, in the amount of time that they will tamper one of the message inside the cumulative message and find the correct nonce for the tampered message and the correct nonce for the cumulative message, the city will be already attacked and destroyed.

In contrast, the generals on the right side just need to append the messages with the given nonces, hash them, and see whether the hash matches or not. Hashing a string is a very easy operation.

Even if PoW solve the Byzantine General's Problem it has a bunch of draws-back:

- It is *extremely inefficient* because of the huge amount of electricity needed to compute all the hashes until it respects the number of leading zeros rule;
- People and organizations that can afford faster and more powerful hash computations have better chance of mining than the other.

Proof of Stake

In Proof of Stake, the creator of a new block is chosen in a deterministic way, depending on its wealth, also defined as stake.
The miners, here called *validators*, will:

- Choose an amount of their coins as initial stake;
- Bet part of their stake in the blocks they think are more likely to be validated;
- If the chosen block gets appended, the miner wins a reward proportional to the initial bet.

The PoS protocol comes with a major draws-back. In the event of a fork, validators can simply put their money in the forked chain without any fear of repercussion at all. No matter what happens, the validator is the only one betting for that chain, so he will always win and have nothing to lose, despite how malicious the actions may be (e.g. a double spending action).

PoW avoided this problem because a malicious node will always mine less blocks compared to the miners in the rest of the network, and if a fork happen, the longest chain will always be chosen, so it is not profitable for a miner to waste so much resource on a block that will be rejected by the network anyway.

This problem is called "Nothing at Stake" and it is solved by the Casper protocol, that is a PoS variant that punishes malicious actors taking all their stakes.

2.1.3 Public Key Infrastructure

A public key infrastructure (PKI) is a set of technologies and solutions that grants secure communications in a network.
The four main elements to PKI are:

Digital Certificates

Public and Private Keys

Certificate Authorities

Certificate Revocation Lists

Digital Certificates A digital certificate is a file that keeps a set of properties related to the holder of the certificate. One of the most common type of certificate is the one compliant with the X.509 standard.
Fig. 2.12 provides an example of X.509 certificate: Mary Morris in the Manufacturing Division of Mitchell Cars in Detroit, Michigan might have a digital certificate with a `SUBJECT` attribute of `C=US, ST=Michigan, L=Detroit, O=Mithcell Cars, OU=Manufacturing, CN=Mary morris /UID=123456`. A digital certificate has a role similar to the an identity card: it provides information and key facts about the holder.

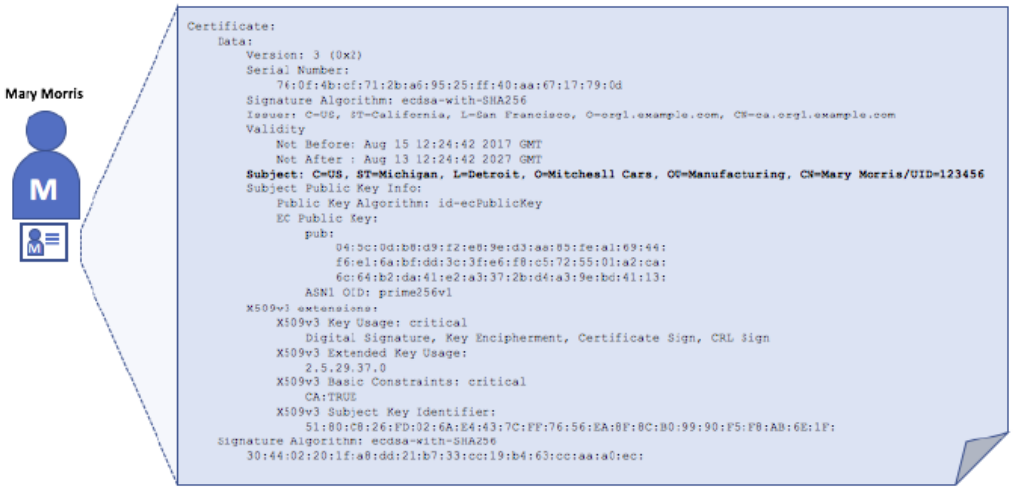


FIGURE 2.12: X.509 Certificate example

An important feature of digital certificates is that they are recorded using cryptography, in a way that any tampering operations will make the certificated invalidated. As long as the other parties trust the certificate issues, known as Certificate Authority (CA), the holder can prove her identity.

Public and Private keys Two key concepts in secure communications are Authentication and message integrity. Authentication means being sure about the identity that wrote a specific message. "Integrity" instead, means that the message cannot have been modified during its transmission.

An important tool to reach authentication are digital signatures that not only allow the sender to digitally sign its messages, but also provide the integrity of the signed message.

To exploit the digital signature mechanism each party must hold two cryptographically connected keys: a private key that is used to create digital signatures on messages and a public key that is available to everybody in the network and works as authentication anchor. The origin and integrity of a message can be verified by making sure that the digital signature is valid according to the public key of the sender.

The cryptographic relationship between the public and the private key of a party makes sure that the signature obtained using a given private key on a message is such that it can only match the corresponding public key and only the same message, as shown in Fig. 2.13.

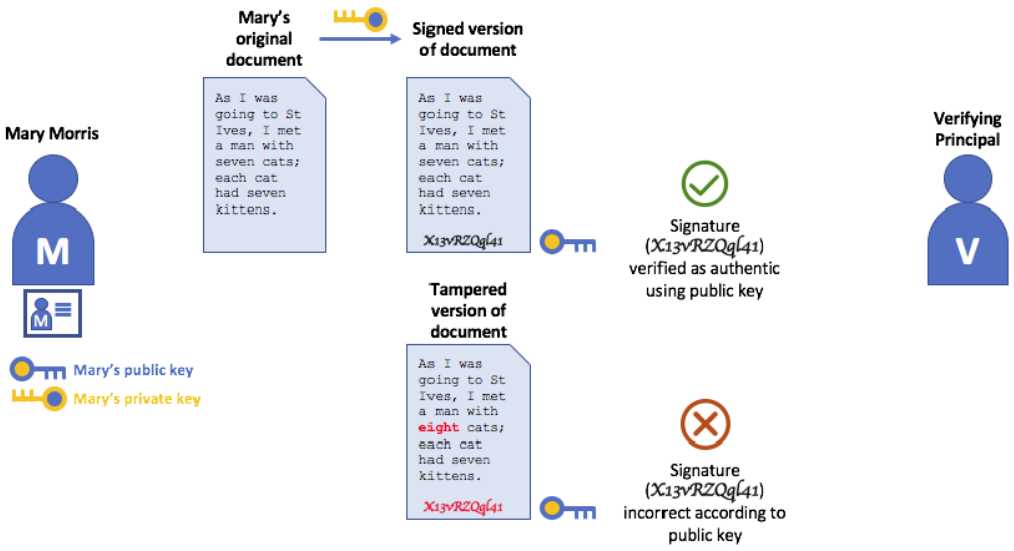


FIGURE 2.13: A party signs the message using her private key. The signature, then, can be verified by anyone using her public key.

Certificate Authorities Certificate Authorities are a common part of internet security protocols. Some of the more popular ones are Symantec, GeoTrust, DigiCert, GoDaddy, and Comodo, among others. As explained in Fig. 2.14, the main role of Certificate Authorities is to dispense certificates to different actors in the network.

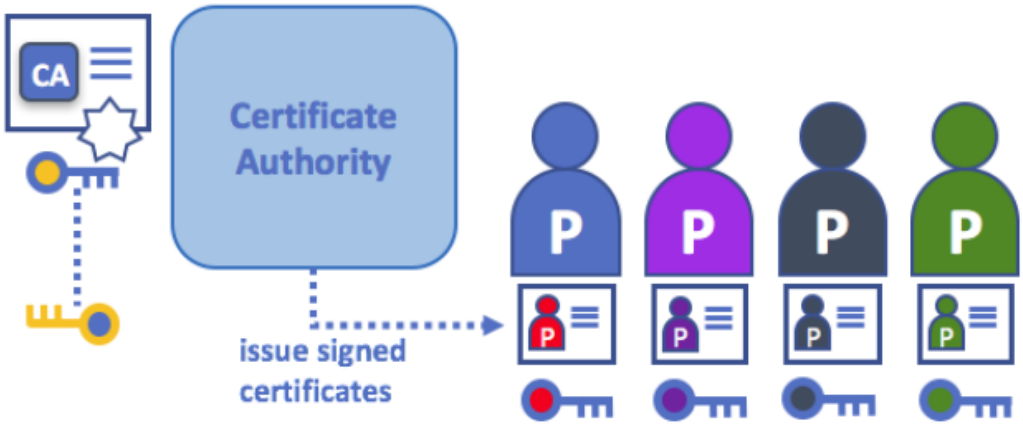


FIGURE 2.14: A Certificate Authority produces certificates for different actors in the network. Every certificate is digitally signed by the CA and bound to the actor with the actor's public key. If a party trusts the CA, it can trust every certificate emitted by that CA.

Certificate Revocation Lists A Certificate Revocation List (CRL) is a list of certificates that a CA has revoked for one reason or another.

As illustrated in Fig. 2.15, when a party wishes to verify another party's identity it can first look the CRL list, to be sure about the validity of that certificate and about the fact that the certificate has not been revoked.

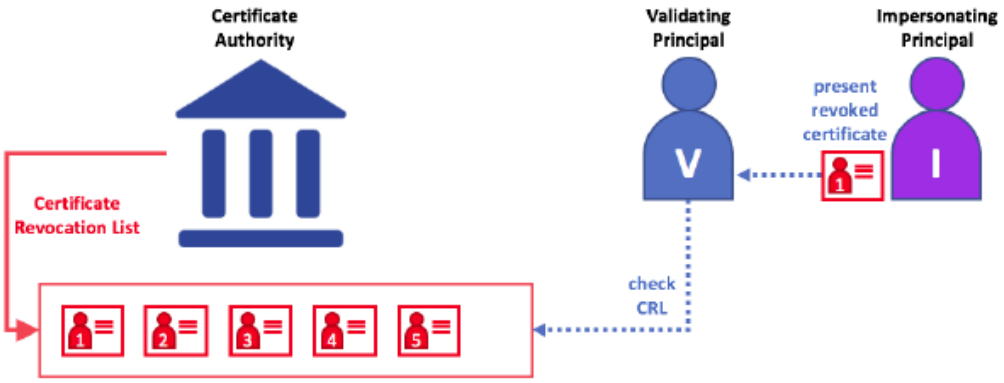


FIGURE 2.15: Thanks to the CRL a party can check whether a certificate is still valid.

2.1.4 Hyperledger Fabric and Ethereum: A Comparison

At time of writing this Thesis the two most mature and famous blockchain frameworks are *Ethereum* and *Hyperledger Fabric*. Hyperledger Fabric provide a *modular* and extendable architecture that can be employed in various industries, from banking and health care over to supply chains. While Ethereum also presents itself as utterly independent of any specific field of application, its components are not pluggable. [8]
The most remarkable differences can be summerized in the following points:

Participation of peers:
With respect to participating to consensus, there are two modes of operation: *permissionless* and *permissioned*. If participation is permissionless, anybody is allowed to participate in the network. This mode is true for Ethereum as a public blockchain. On the other hand, if participation is permissioned, participants are selected in advance and access to the network is restricted to these only. This is true for Fabric.

Consensus:
In the current implementation of Ethereum, the *consensus* mechanism is established by mining based on the proof-of-work (PoW) scheme. All participants have to agree upon a common ledger and all participants have access to all entries ever recorded. The consequences are that PoW unfavorably affects the performance of transactions processing. Concerning the data stored on the ledger, even though records are anonymized, they are nevertheless accessible to all participants, which is problematic for applications that require a higher degree of privacy.
Fabric's understanding of consensus is broad and encompasses the whole transaction flow, starting from proposing a transaction to the network to committing it to the ledger. Furthermore, nodes assume *different roles* and tasks in the process of reaching consensus. This contrasts to Ethereum where roles and tasks of nodes participating in reaching consensus are identical.
With Fabric, the algorithm employed is "pluggable", meaning that depending on application-specific requirements various algorithms can be used.

Smart Contracts:
Smart Contracts can be written in Go or Java for Fabric and in Solidity for

Ethereum. This means the Fabric offers widespread programming languages while Ethereum requires developers to learn Solidity.

Built-in currency:
Another difference lies on the fact that Ethereum has a build-in cryptocurrency called Ether, that is used to pay the transaction fees and to reward the miners. Fabric does not require a build-in cryptocurrency as consensus is not reached via mining.

To sum up they both are highly flexible, but in different aspects.

Ethereum's powerful smart contracts engine makes it a generic platform for literally any kind of application. However, Ethereum's permissionless mode of operation and its total transparency comes at the cost of performance scalability and privacy. Fabric solves performance scalability and privacy issues by permissioned mode of operation. Further, the modular architecture allows Fabric to be customized to a multitude of applications.

2.2 Hyperledger Fabric

This section provides a summary of the Hyperldeger Fabric official documentation [1]. Hyperledger Fabric is a framework for distributed ledger solutions under the Linux Foundation license.

Hyperledger Fabric provides the following blockchain network functionalities:

Identity management Differently from any other blockchain system Hyperledger Fabric is private and permissioned. Differently from an open permissionless system where everybody can participate in the network (requiring protocols like "proof of work" to validate transactions and secure the network), Hyperledger Fabric enrolls new members of the network through a trusted Membership Service Provider (MSP).

Hyperledger Fabric uses public Key Infrastructure to produce cryptographic certificates which are tied to organizations, network components, and end users or client applications. Moreover, an access control lists can be provided to add additional layers of permission.

Efficient processing In Hyperledger Fabric every node has one or more roles. To enhance concurrency and parallelism on the network, transaction commitment and transaction execution are kept separated.

Chaincode functionality External application that needs to interact with the ledger can invoke Hyperledger Fabric smart contracts, also called chaincode. Smart contracts can be written in different programming languages, such as Go and Node.

Privacy and confidentiality Hyperledger Fabric also provides the opportunity to create channels. A ledger exists in the scope of a channel. In this way a group of participants can create a separate ledger of transactions.

Modular design Hyperledger Fabric architecture offers several pluggable options for what concerning consensus protocols and formats.

2.2.1 Fabric network architecture

A Hyperledger Fabric network consists of:

Peer nodes

Ordering service(s)

Channel(s)

Fabric Certificate Authorities

Ledgers (one per channel - comprised of the blockchain and the state database)

Smart contract(s) (aka chaincode)

The users of the network services are:

- Clients of Blockchain network administrators
- Client applications owned by organizations

Fig 2.16 represents a possible final state of a Hyperledger Fabric network.

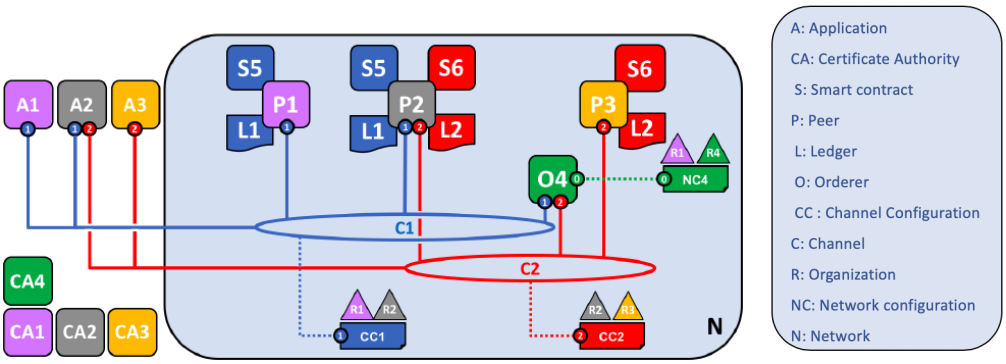


FIGURE 2.16: A complete Fabric sample network

Suppose that four organizations, R1, R2, R3 and R4 want to set up a Hyperledger Fabric network and R4 has been chosen as the organization with the power to set up the first version of the network. R4 does not want to make transactions on this network, while R1 and R2 need a private channel between them, as well as R2 and R3.
Organization R1 owns an application to interact with the network and make transactions within channel C1.
Similarly, R2 can perform transactions both in channel C1 and C2, while R3 can do the same on channel C2.
Peer node P1 keeps a copy of the ledger L1 bound to the channel C1 and a chaincode S5 can be invoked from P1, so that client application A1 can read and update the ledger through peer node P1.

Peer node P2 has both a copy of the ledger L1 and a copy of ledger L2, while P3 just keeps a copy of the ledger L2.

The network follows the policy rules defined in the network configuration NC4, chosen by R1 and R4.
Channel C1 follows the policy rules defined in the channel configuration CC1 chosen by R1 and R2.

Channel C2 follows the policy rules defined in channel configuration CC2 chosen by R2 and R3.

Moreover, an ordering service O4 orders the transactions made in the channels C1 and C2, into blocks for distribution. Every organizations owns a Certificate Authority.

Creating the network As shown in Fig 2.17, the first step to form a network is to start an orderer. In this sample network N, the ordering service includes just one node, O4, following the policy described in the network configuration NC4. Certificate Authority CA4 creates identities to the nodes of the organization R4. Certificates made by CAs are very important, since they are used to produce digital signatures in the transactions to represent that an organization endorses the transaction result. Since CAs are very important, Hyperledger Fabric provides with a built-in one (called *Fabric-CA*, though organizations will choose to use their own CA.

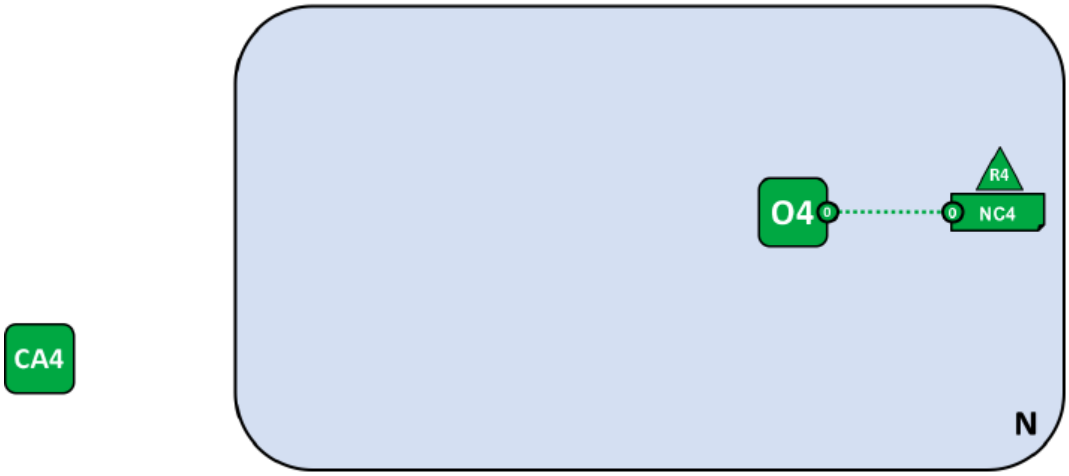


FIGURE 2.17: Step 1: Create the Fabric sample network

Adding Network Administrators Initially only R4 had the administrative rights on NC4. Now, organization R4 can add organization R1 as another administrator. At this point R1 and R4 have the same rights over the network configuration.
Certificate authority CA1 must now be added to issue certificates for users from the organization R1.

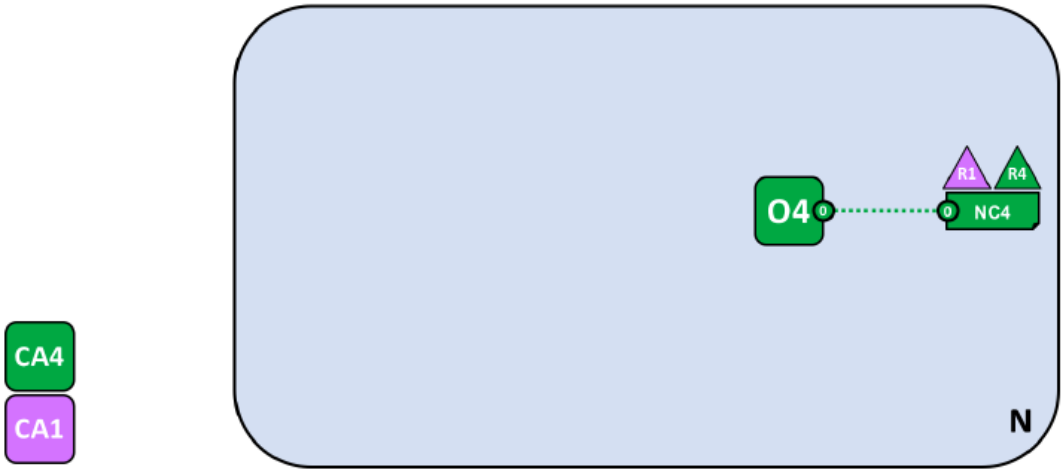


FIGURE 2.18: Step 2: Add network administrators

Defining a consortium Figure 2.19 shows R1 and R2 creating a new consortium X1 between each other. A consortium between different organizations must be created every time those organization need to transact with one another. CA2 must now be added to issue certificates for users from R2.

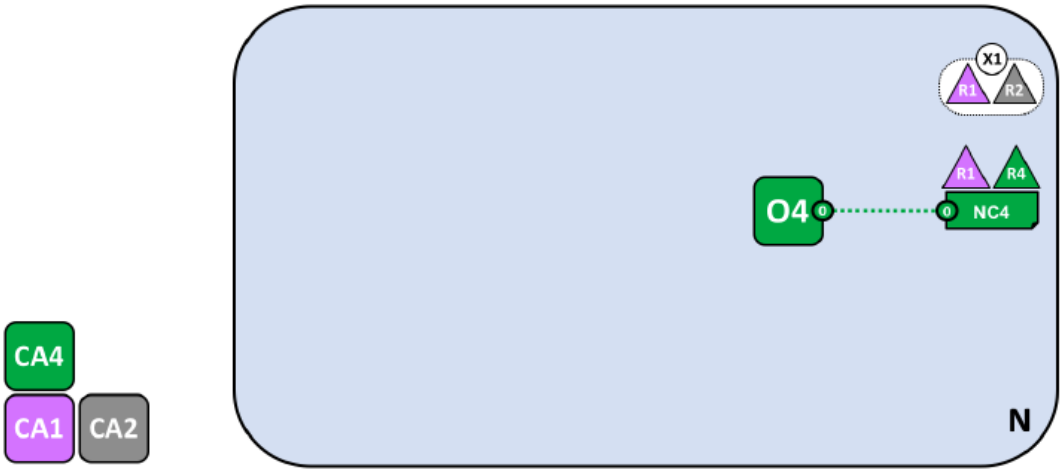


FIGURE 2.19: Step 3: Define a consortium

Creating a channel for a consortium If the organizations of a consortium need to communicate to each other, a channel must be created. A network can host multiple channels.
In this sample R1 and R2 (consortium X1) share a channel C1. The channel is under the channel configuration CC1, managed by R1 and R2 who have equal rights over C1.

Using channels allows to share the network infrastructure while maintaining data and communications privacy.

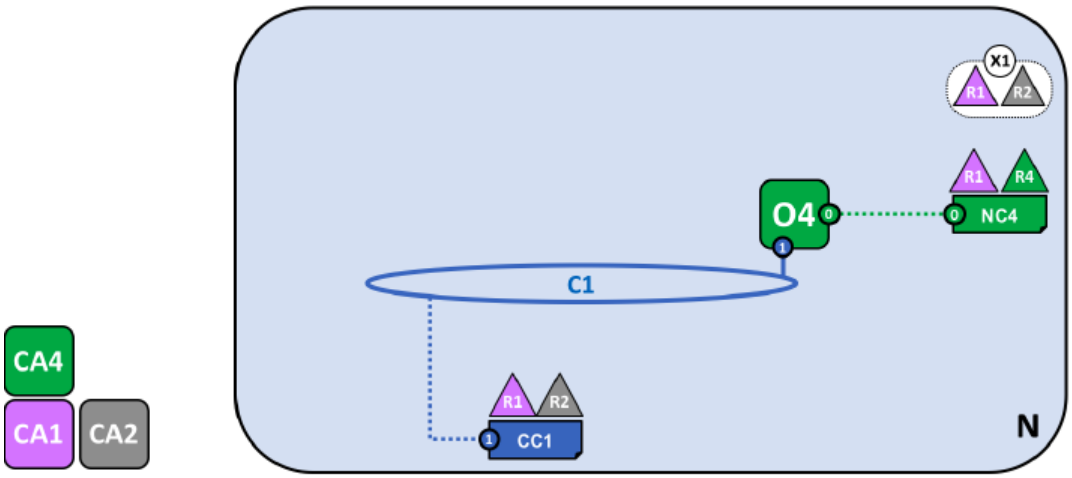


FIGURE 2.20: Step 4: Create a channel

Peers and Ledgers As showns in fig. 2.21 two new components,a peer node P1 and a ledger L1 are now added to the sample network.
P1 is now part of the channel C1 and holds a copy of the ledger L1.
L1 is physically hosted on P1, but logically hosted on the channel C1.

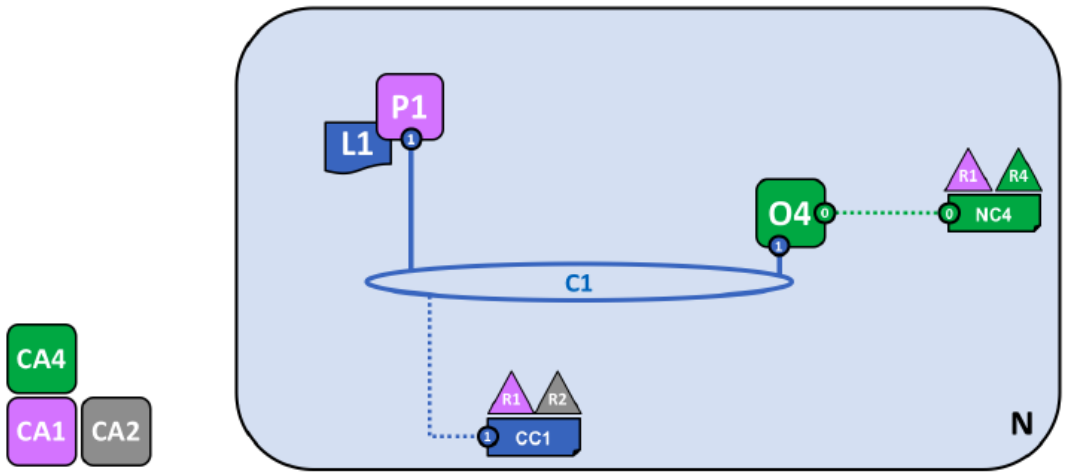


FIGURE 2.21: Step 5: Adding peers and ledgers

Applications and Smart Contracts Now, client applications can interact with the ledger for query and/or update transactions.
As shown in fig. 2.22, thanks to the smart contract S5, installed on P1, client application A1 can access the ledger through peer node P1.
It may look like A1 can directly access L1, however, the ledger can be accesses only through a program called a smart contract chaincode, S5 that provides a well-defined set of ways by which the ledger L1 can be queried or updated.
Smart contracts must have been *installed* and *instantiated*.

S5 must be installed not only on P1, but also on the channel C1 so that all the other components connected to channel C1 becomes aware of it.

Components in the channel, are still unable to see S5 program logic. This remains private to those nodes who have installed it, in this case P1.

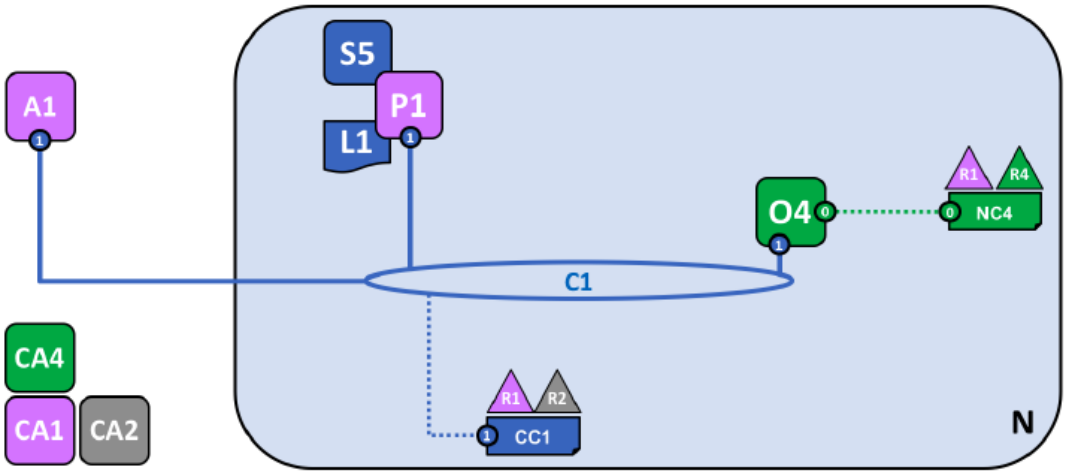


FIGURE 2.22: Step 6: Adding client applications and smart contracts

Complete the network Fig. 2.23 shows the next phase of network developments: adding organization R2 to the network. R2 added peer node P2, which hosts a copy of ledger L1, and chaincode S5. Thanks to application A2, P2 can now join channel C1. Similarly to A1, also A2 can now interact with the ledger L1.

At this point, the two organizations R1 and R2 can fully transact with each other through channel C1.

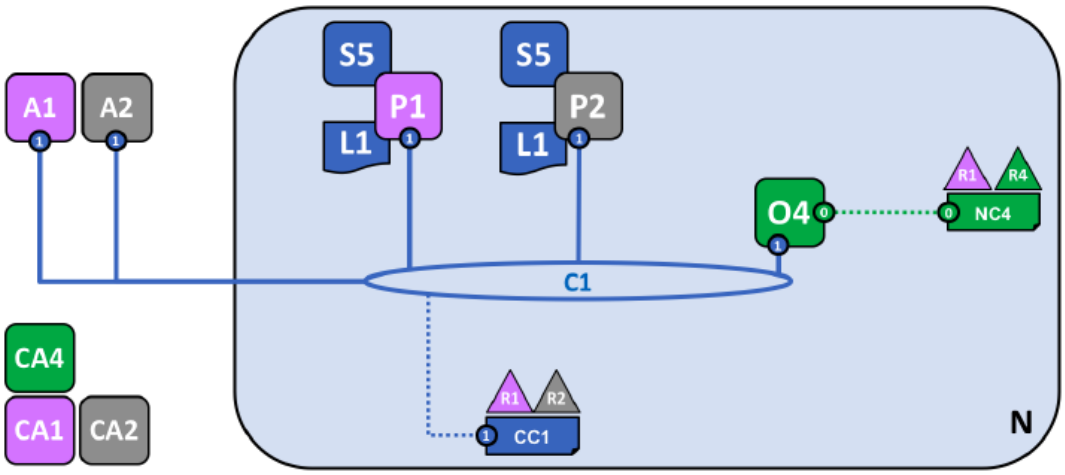


FIGURE 2.23: Step 7: Completing the network

Since also R2 and R3 need to transact each other, a network administrator (from R1 or R4) must define a new consortium, X2, made of R2 and R3 and a new channel C2 for the new consortium, together with a channel configuration CC2. This architecture is described in Fig. 2.16. Peer P3, representing the organization R3, must be added. P3 clients can use the application A3 to interact with the ledger L2.

Peer node P2 is both a member of channel C1 and channel C2, so it must have installed both smart contract S5 and smart contract S6. This is a very powerful feature of Hyperledger Fabric - thanks to channels organizations can share the same

network architecture, and communicate privately at the same time.

It is also possible to have more than one Ordering service, as described in Fig. 2.24.

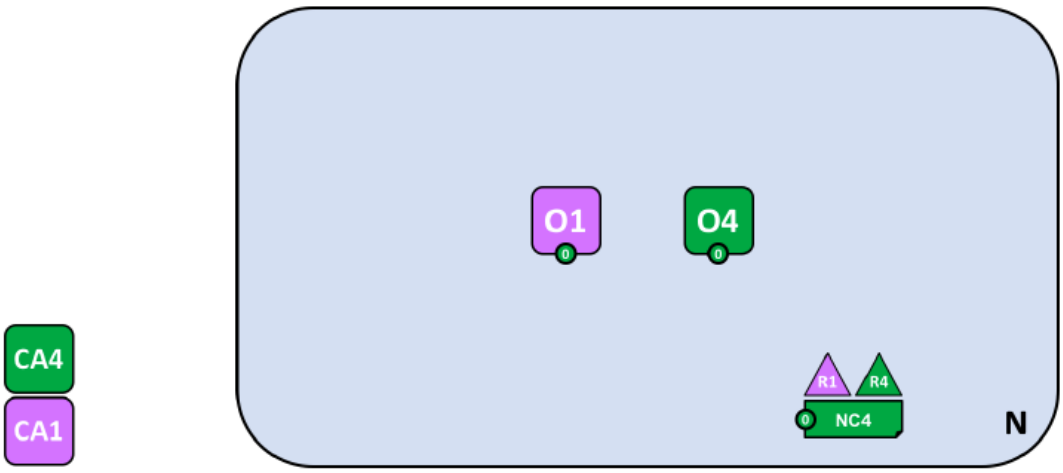


FIGURE 2.24: Ordering service: a de-centralized example

A blockchain application example, provided by IBM [3] is a web-based blockchain insurance application. The app will have four participants, or peers:

Shop

Repair shop

Insurance

Police

The Shop peer sells a product to a consumer, that can buy additional insurance services thanks to the insurance peer, i.e., the organization that provides the insurance services for a product, and that is responsible for receiving the claims. The Police participates to the network with a peer responsible for validating the accident or theft claims. Once the accident has been verified, a Repair shop peer is responsible for repairs of the product. Fig. 2.25 better defines the network architecture.

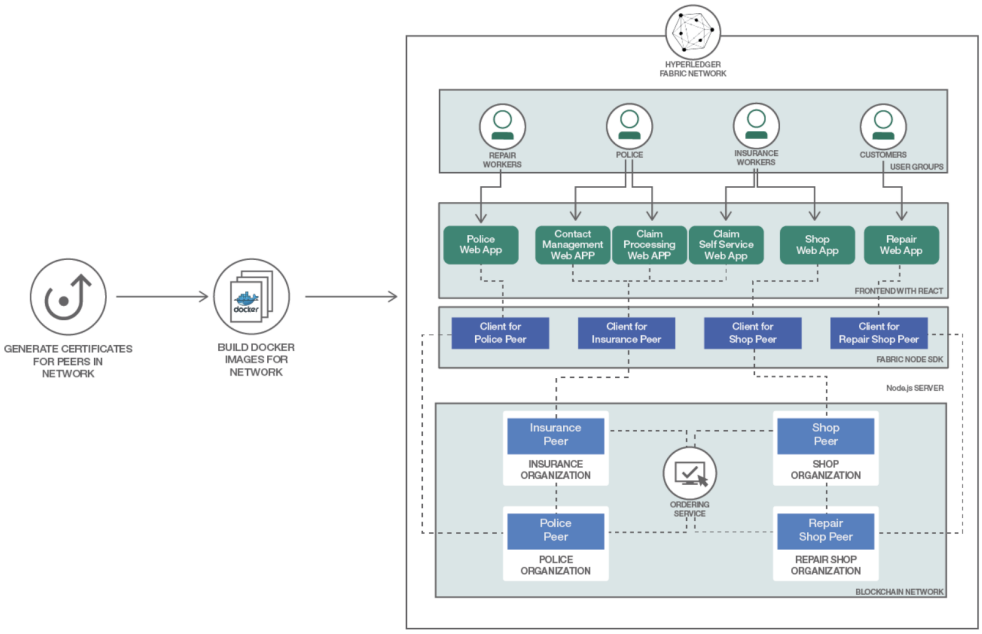


FIGURE 2.25: IBM Insurance application work-flow

2.2.2 Types of peers

In Hyperledger Fabric peers can have different roles:

Committing peer. It is a peer that is in charge of committing blocks of validated transactions. Every peer that has a copy of the ledger in a channel is a committing peer.

Endorsing peer. It is defined by policy as specific node that executes smart contract transactions in simulation and returns a proposal response (endorsement) to the client application.
Defining the endorsement policy for a smart contract means defining the organizations whose peers are required to digitally sign a transaction before committing it to the ledger.

2.2.3 Identity

Actors in a blockchain network can be orderers, peers, administrators, client applications and more. Every actor must have a digital identity in an X.509 digital certificate. The identities define the permissions over resources of every actors in the blockchain network.

The built-in Fabric Certificate Authority — known as Fabric CA manages digital identities of Fabric members that have the form of X.509 certificates.

Membership Service Provider (MSP) Suppose that at the checkout in a store only some cards are accepted, e.g. Mastercard and Visa. You may have another card, for example American Express, that is valid and that contains sufficient money, however that card type is not accepted.

FIGURE 2.26: PKI and MSPs have a complementary role, PKI provides identities while MSP defines which of these identities are members of the Fabric network.

As described in Fig. 2.26 PKI and MSPs work together in a complementary way. A PKI issues many different types of verifiable identities. While the MSP, defines which of these identities are the trusted participants of the store payment network. MSPs turn verifiable identities into the members of a blockchain network.

The Membership Service Provider defines which Root CAs and Intermediate CAs are trusted to define the participants of a given network.

Local and Channel MSPs There are two places in which MSPs appears within a blockchain network: channel configuration (channel MSPs), and locally on the actor itself (local MSP). Local MSPs (present in users, peers and orderes) describes the rights for that node.

Channel MSPs, instead, describes administrative and participatory rights at the channel level.

2.2.4 Basic transactions work flow

There are two main type of transactions:

Ledger-query: they involve a simple three-step dialogue between an application and a peer.

Ledger-update: They require two extra steps.

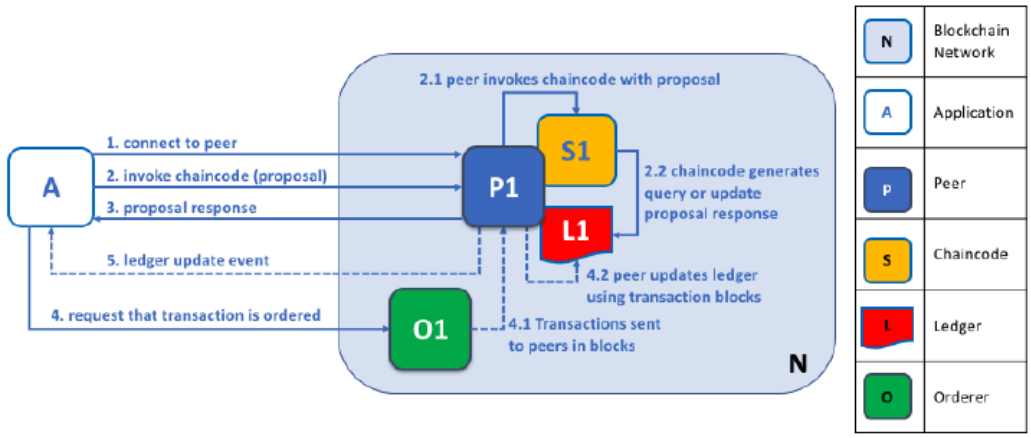


FIGURE 2.27: Thanks to Peer P1, application A can invoke the smart contract S1 for a given transaction. S1 produces a simulation of the transaction and P1 sends back to A the transaction proposal responses. A ledger-query ends here, while for a ledger-update A sends the transaction together with the transaction proposal responses to O1, that after creating a new block, sends the block to every committing peer in the network.

Phase 1: Proposal Phase 1 of the transaction work-flow only involve peers, not orderers. In phase 1, applications send a transaction proposal to every peer included in the endorsment policy. Each of these peers independently executes the transaction and sends back to the application a transaction proposal response. These peers do not apply any update to the ledger, they just simulate the transaction to compute the read and write set. Phase 1 ends with the application having all the transaction proposal responses required by the endorsment policy.

In the case in which peers send back to the application different and inconsistent transaction proposal responses, the application just asks for a more up-to-date response.

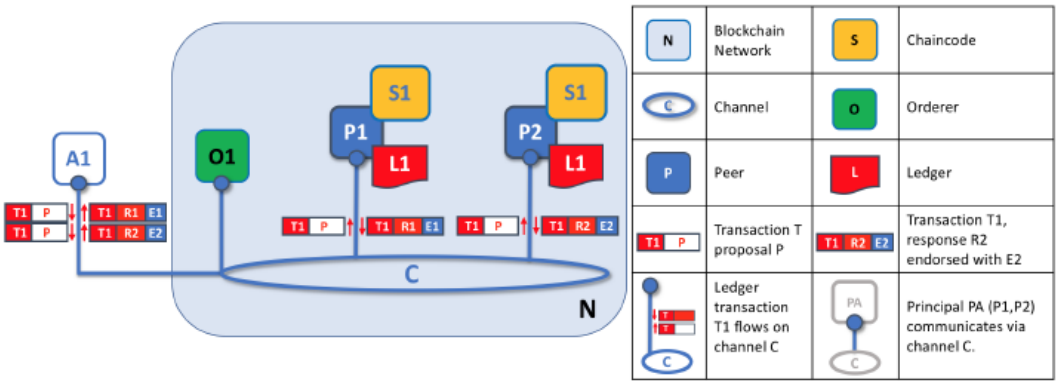


FIGURE 2.28: Transaction work-flow: Phase 1

Phase 2: Packaging Phase 2 of the transaction work-flow, the packaging, mainly involves the orderer. Once the application has all the required transaction proposal responses, it sends the transaction proposal together with all the endorsments to the

orderer. Once the orderer collects many transactions, it orders them and create a block ready to be sent back to all the committing peers.

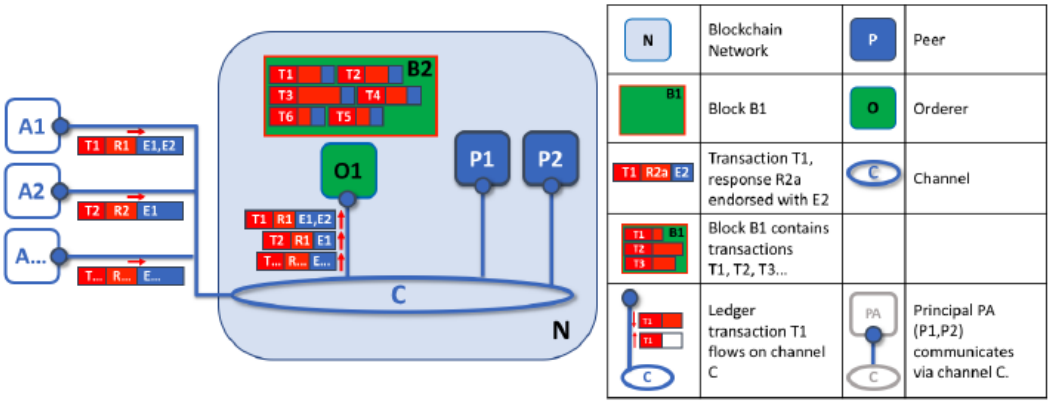


FIGURE 2.29: Transaction work-flow: Phase 2

The order in which the orderer sort the transactions may not be the same as the arrival order. What is important is that there is an order, and that the order will be the same for every commmitting peer, instead of what the order is.

Phase 3: Validation In phase 3, the validation, the orderer sends the prepared block to every committing peer in the network. Each peer, independently process every transaction in the block, checking if the transaction owns all the required endorsments and applies the update to the ledger.

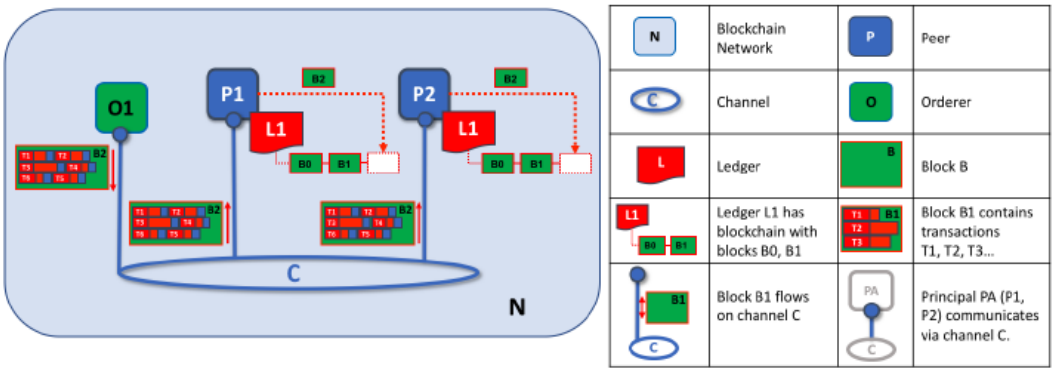


FIGURE 2.30: Transaction work-flow: Phase 3

The overall Hyperledger Fabric transaction flow is shown in Fig. 2.31.

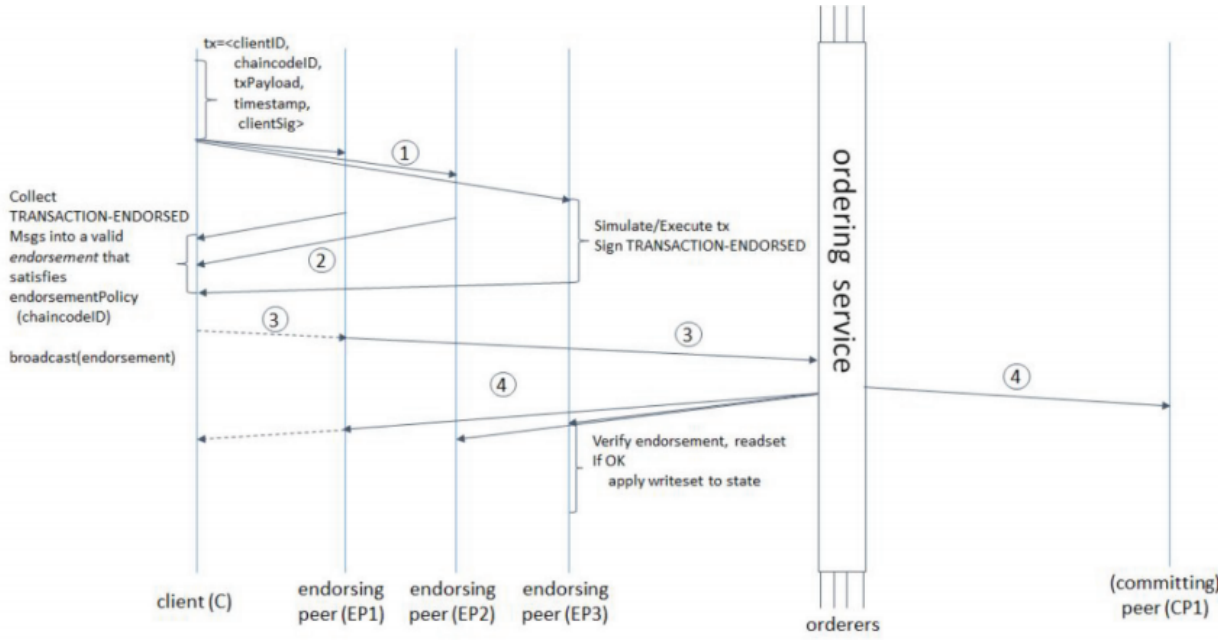


FIGURE 2.31: Transaction work-flow

In all three phases: endorsment, ordering and validation Hyperledger Fabric supports pluggable consensus service.

Multiple ordering plugins are being developed currently, including BFT Smart, Simplified Byzantine Fault Tolerance (SBFT), Honey Badger of BFT, etc. For Fabric v1, Apache Kafka is provided out-of-the-box as a reference implementation. The application use-cases and its fault tolerance model should determine which plugin to use.

2.2.5 Ledger

A blockchain ledger is made of two different parts:

The world state - usually defined as key-value pairs, it contains the current values of the database.

The blockchain - a read-append-only transaction log.

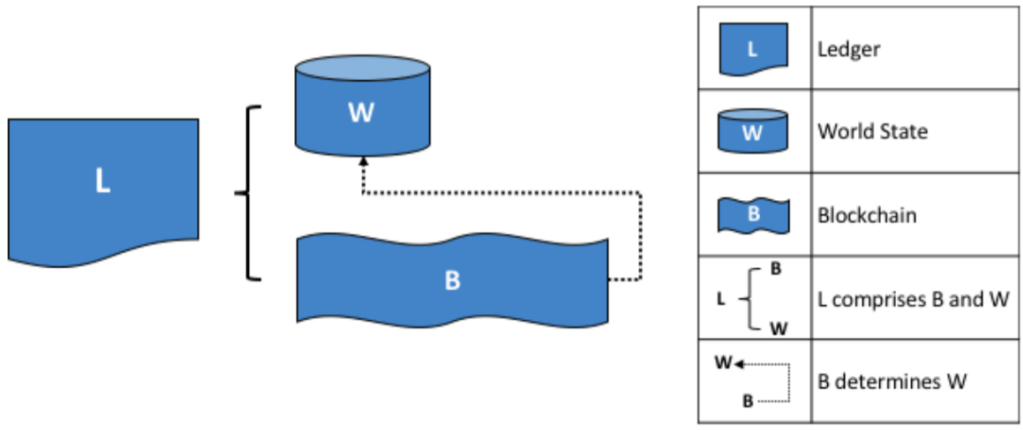


FIGURE 2.32: Ledger L is made of blockchain B and World State W. Blockchain B implicitly defines the World State W.

World State The world state contains the current values of each record in the database.
The world state can be currently implemented as LevelDB and CouchDB. As described id Fig. **??** a state also includes a version number. The version number of a state is a very important feature. Indeed it is incremented every time the state changes, so that whenever an update must be applied to the state - we can check if the version matches the version of when the transaction was created.

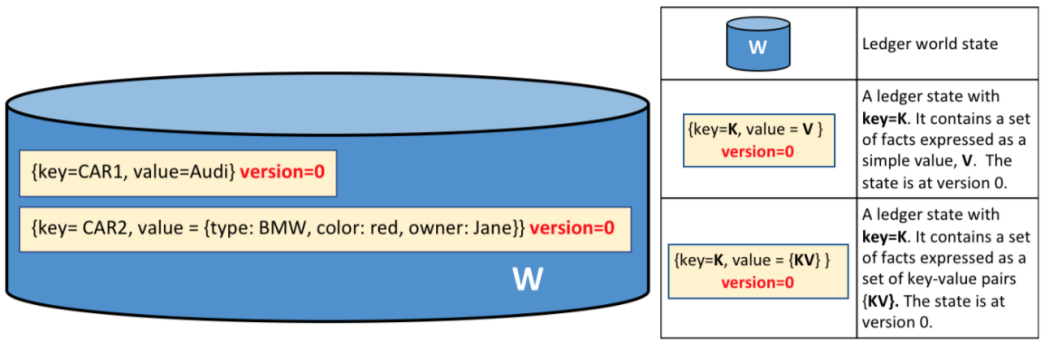


FIGURE 2.33: Two examples of world state.

Blockchain The blockchain is a transaction log, made of linked blocks. Blocks are made of batches of transactions, each of which can be a ledger-query or a ledger-update.

Each block's header contains the hash of the block together with the hash of the previous block, so that each block is linked to the previous one.

The Blockchain is usually implement as a file. This is a quite obvious design choice since update operation are very unusual while append operations are the most frequent ones.

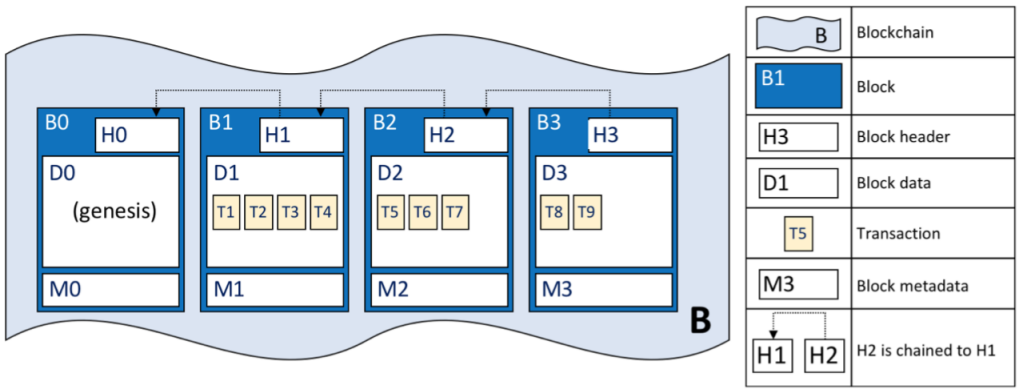


FIGURE 2.34: Blockchain B is made of blocks B0, B1, B2, B3, where B0 is the genesis block.

The above diagram shows that the first block does not contain any transaction. Instead, it containts a configuration and initial state of the channel.

Blocks Each block has three main sections:

- Block Header
  A section that includes three different fields:
  - Previous Block Hash: The hash of the previous block in the chain
  - Block number: a sequence number for the blocks
  - Current Block Hash: The hash of the current block
- Block Data
  This section includes a list of ordered transactions
- Block Metadata
  This section includes metadata such as: a timestamp of when the block was written, the certificate, public key and signature of the block writer.

Transactions The block data section includes a set of ordered transactions.

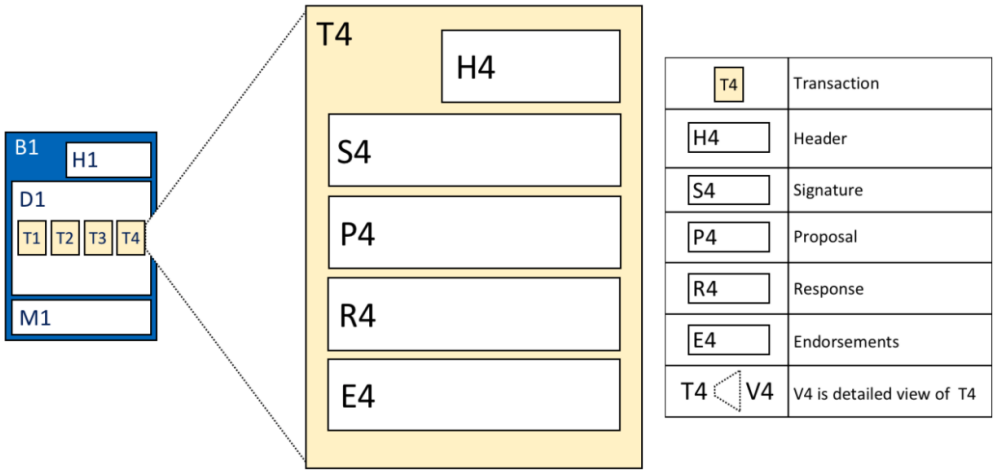


FIGURE 2.35: Transaction T4 is made of the header, the signature, the proposal, the response and the endorsments.

The mains fields in a transaction are:

- Header
  The header includes some metadata such as chaincode name and version.
- Signature
  The signature is generated using the application's private key to generate it, and works as a proof that the transaction has not been tampered.
- Proposal
  The proposal contains the list of input parameters given by the application to execute a given transaction in the chaincode
- Response
  The response includes the output of the chaincode, made of a Read Write set (RW-set).
- Endorsements
  The endorsments are a list of transaction proposal responses computes from the peers included in the endorsment policy.

2.3 Hyperledger Composer

Hyperledger Composer is an open-source project under the Linux Foundation umbrella. Hyperledger Composer enables architects and developers to quickly create Blockchain solutions with REST APIs that expose the business logic to web or mobile applications, as well as integrating blockchain with existing enterprise systems of record.
Hyperledger Composer supports the existing Hyperledger Fabric blockchain infrastructure and runtime.

Hyperledger Composer consists of:

- A modelling language called CTO (an homage to the original project name, Concerto);

- A user interface called Hyperledger Composer Playground for rapid configuration, deployment, and testing of a business network;
- Command-Line Interface (CLI) tools for integrating business networks modeled using Hyperledger Composer with a running instance of the Hyperledger Fabric blockchain network.

Hyperledger Composer allows to quickly model a business network made of assets and transactions related to them. Assets are tangible or intangible goods, services, or property. Transactions interact with assets. Business networks also include the participants who interact with them, each of which can be associated with a unique identity.

Hyperledger Composer allows to define queries within a business network. Queries are used to return data about the blockchain world-state and can include variable parameters for simple customization.

2.3.1 Business Network Definition

The Business Network Definition is a key concept of the Hyperledger Composer programming model.
Business Network Definitions are composed of:

- A set of model files;
- A set of JavaScript files;
- An Access Control file.

The model files define the business domain for a business network. They contain the definitions of assets, participants and transactions.
The JavaScript files contain transaction processor functions that can run on top of a Hyperledger Fabric blockchain network.
The permissions for the business network are expressed in an optional `permissions.acl` file. Once defined, a Business Network Definition can be packaged into a business network archive file (.bna). This process is explained in Fig. 2.36.

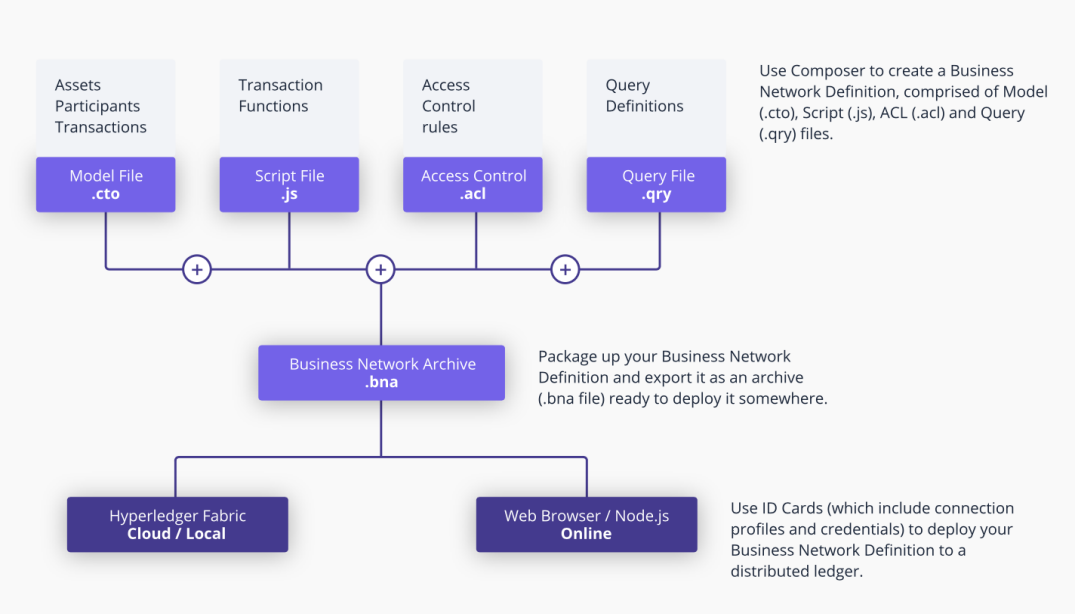


FIGURE 2.36: Hyperledger Composer solution design

Once the `.bna` file is ready, it can be deployed to the Hyperledger Fabric blockchain.

Chapter 3

Proposed Solution

3.1 Business Network Definition

3.1.1 Model

The proposed solution has been developed according to the architecture described in the Figure 3.1. Five different type of participants have been defined: campaign, donor, student, mentor and vendor.

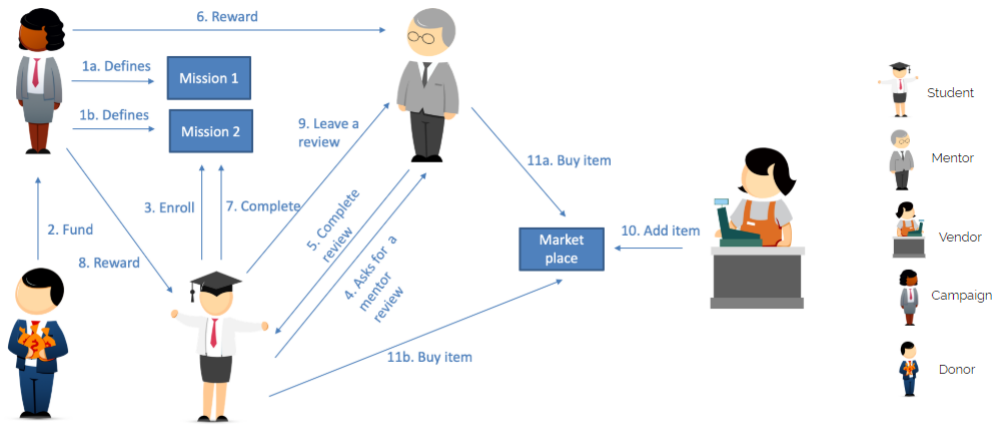


FIGURE 3.1: Business Network Architecture

Since they have a lot of attributes in common, an `abstract Business` class has been defined. All the other participants inherit the common attributes from it.

```
/*
* An abstract participant type in this business network
*/
abstract participant Business identified by email {
  o String email
  o String firstName
  o String lastName
  o Address address optional
  o Integer educoinBalance default=0 optional
}
/*
A campaign is also a participant, since it can transfer and own educoins
 */
participant Campaign extends Business {
  o String campaignName
  o String campaignDescription
```

```
  o Integer fundingGoal default=0
  o Boolean funded // this means that the funding goal has been reached
  o Boolean completed // this means that all the activities have been added
}
/**
* A donor is a type of participant in the network
*/
participant Donor extends Business {
}
/**
* A student is a type of participant in the network
*/
participant Student extends Business {
}
/**
* A mentor is a type of participant in the network
*/
participant Mentor extends Business {
  o Double reviewSum default=0.0 optional
  o Double reviewCount default=0.0 optional
  o Review[] reviews optional
}
/**
* A vendor is a type of participant in the network
*/
participant Vendor extends Business {
}
```

As soon as a Campaign is created, it can start defining its missions. A `mission` is a set of education activities that a Student can decide to engage with.

```
/**
* The activities to be completed inside a mission
 */
concept Activity{
  o String name
  o String description
  o Integer educoin default = 0
}
/**
* A mission to be complete from mentors/students to earn EduCoin
*/
asset Mission identified by missionId {
  o String missionId
  o String missionName
  o DateTime dueDate
  o String missionDescription optional
  o Integer educoin default = 0 // = to the sum of the educoin for each
      assignment
  o Integer bonusEducoin
  o Integer maxStudents
  o Integer mentorFare
  o Integer currentStudents default=0
  o Activity[] activities
  --> Campaign campaign
}
```

Every `mission` can host a maximum number of students (`maxStudents`) and must be completed by a certain `dueDate`.

What a Campaign expects from a student is to correctly address and complete at least one of the mission activities and to submit at least one document containing the result produced by the student.

For the Campaign to start it must define a funding goal, i.e. the total amount of Educoin (EDC) needed to start and complete the campaign.
In the definition of the funding goal three main assumptions have been made:

1. The first student who successfully completes all the activities of a mission receive an extra Educoin bonus.
2. Every student can ask for a review from one of the mentors registered to the platform. Mentors are rewarded for every review they do through Educoin.
3. For the sake of simplicity, for a given mission the student can ask for a review by the mentor just once.

Following these assumptions the founding goal for a given Campaign is computed as:

$$\sum_{m \in M} [maxS_m * (\sum_{a \in A} e_a + mf_m) + b_m]$$

where:

M = set of missions for a given campaign;

A = set of activities for a given mission;

$maxS_m$ = max number of students for mission m;

e_a = educoin earned by completing activity a;

mf_m = mentor fare for mission m;

$bonus_m$ = bonus educoin earned for mission m;

Once all the missions have been defined, the funding goal can be computed and the Campaign is ready to be funded.

As soon as the Campaign receives all the required fund, it can officially start and students will be able to enroll the missions offered by that campaign.
The contract between a Student and Mission is recorded in a `MissionContract`.

```
asset MissionContract identified by missionContractId {
  o String missionContractId
  o Boolean completed default=false
  o Boolean mentorUsed default=false
  o DateTime dueDate
  o DateTime reviewDate optional
  o DateTime mentorSubmissionDate optional
  o DateTime submissionDate optional
  o Integer earnedEducoin default=0
  o Boolean winner default=false
  o Boolean[] completedActivities optional
  o String[] filenames optional
  o String[] attachments optional
  --> Mission mission
  --> Student student
  --> Mentor mentor optional
}
```

The mission contract represents the contract between the Student `student` and the Mission `mission`. The mission contract records not only whether or not the mentor has already been used through the boolean variable `mentorUsed` but also which mentor has been chosen. This information allows the smart contract to reward also the mentor once the review has been completed.

The mission contract keeps track of four different dates:

`dueDate`: the date by which the contract is valid. If the student completes the mission after this data he will not be rewarded.

`submissionDate`: the date in which the Student has submitted the final document to the Mission.

`mentorSubmissionDate`: the date in which the Student has submitted the document to the Mentor.

`reviewDate`: the date in which the Campaign has reviewed the document submitted by the student and decided which activities have been successfully completed.

The mission contract also records the educoin earned by the Student (`earnedEducoin`) and whether or not the Student was the winner for that mission.

Once the due date passed, the Campaign can evaluate the student's work and decide which activities have been successfully completed; the mission contract records this information in `completedActivities`. Finally the mission contract keeps track of all the attachments (both the ones submitted by the student and the ones submitted by the mentor) together with their file names.

Once the student uses the mentor review service she has the possibility to leave a review to the mentor. This is a classic review, where she can describe how the mentor behaved, whether or not his corrections have been useful, etc. and finally give him a score, so that other students can benefit from this information.
Indeed, before choosing a mentor, students can access a list of mentors together with their reviews and their average score.

```
concept Review {
  o String title
  o String description
  o Double score
}
```

Last but not least, a Vendor can upload the items' descriptions to the B4E Market Place, so that students and mentors can buy them using their Educoins. The Educoin price must be previously agreed with the platform responsible (that in this case can be the Education Global Practice inside the World Bank Group). Currently, for the sake of simplicity, only a network administrator can upload items to the market place.

```
/**
* Items to be sold in the market place
 */
asset Item identified by itemId{
  o String itemId
  o String name
```

```
  o String description
  o Integer cost
  --> Business owner
}
```

An `Item` contains simple attributes such as the name, the description, the cost and the owner. We can notice that the `owner` type is `Business`. However the access control list ensures that only a Student, a Mentor or a Vendor can own an item, thus preventing donors or campaigns to perform transactions inside the market place.

The complete `model.cto` file can be found in Appendix A.1.

3.1.2 Limitations

Since Hyperledger Fabric has not a built-in currency, cryptocurrency transactions are not as straightforward as they would have been using Ethereum. The Educoin balance of each participant (Campaign, Student, Mentor, Donor and Vendor) is registered exactly as an attribute in a table of a database.

When a donor wants to fund a campaign he is already supposed to have Educoin, but how can a donor buy Educoin, e.g. exchanging USD dollars for them? In a similar way, when the vendor sells his items, he receives Educoin, but sooner or later he may wish to exchange them for the local currency.

Since Educoin is not properly a cryptocurrency, but more an attribute in a table, payments from donors to the platform and from the platform to the vendors must be managed off-chain, e.g. through Paypal or other payments method. However, payment receipts can of course be saved on the blockchain.

You may notice that also students and mentors may wish to exchange the Educoins for the FIAT currency, but since, differently to donors and vendors, they can use their coins inside the marketplace, the FIAT exchange is not a real requirement.

3.1.3 Logic

The different transactions that can be performed by the participants inside the B4E network are:

`AddMissionToCampaign`: allows a Campaign to define and add a new Mission in its missions list.

`FundCampaign`: allows a Donor to fund a chosen Campaign of a given amount.

`EnrollStudentToMission`: allows a Student to engage with a given mission. Here is when the `MissionContract` between the Student and the Mission is created.

`StudentAskForMentor`: allows the Student to select a mentor from the list and ask for his help. The student must submit at least one document which will be visible to the mentor. The mentor is now added to the `MissionContract`.

`MentorReivew`: allows the Mentor to submit a new document. It can be both the same document upload by the student, with some modifications or a completely new document.

`StudentLeaveReview`: allows the Student to leave a review to the Mentor she used, describing and evaluating his work and behavior.

`StudentSubmitMission`: allows the Student to submit the final document to the Mission. She can do more than one submission before the due date, but in the final evaluation the Campaign will just consider the last one.

`StudentCompleteMission`: allows the Campaign to record which activities have been successfully completed. The amount of educoin earned by the student will be automatically computed accordingly.

`AddItem`: allows a Vendor to create and add a new item to the market place.

`BuyItem`: allows a Student or a Mentor to buy an item from the market place.

`AddMissionToCampaign` When a Campaign logs into the platform it can start defining its own missions. Every time a new mission is created the Campaign must specify whether this is the last mission or not (`completeCampaign = true`). This information allows the system to know when the funding goal can be computed and the campaign can start to be funded from the donors.

There are also other information required by a `AddMissionToCampaign` transaction: the name (`missionName`), a brief description (`missionDescription`), the date by which it must be completed by the students (`dueDate`), the bonus that the first student who completes all the activities of the mission will earn (`bonusEducoin`). The Campaign must also specify the list of activities that students can complete (`activities`), together with their names, descriptions and Educoin prizes, the maximum number of students that can engage with this mission (`maxStudents`) and the mentor fare for this mission, i.e. the amount of Educoin that a mentor will earn by doing a review for a student in this mission (`mentorFare`).

```
transaction AddMissionToCampaign {
  o String missionName
  o String missionDescription
  o DateTime dueDate
  o Boolean completeCampaign
  o Integer bonusEducoin
  o Integer maxStudents
  o Integer mentorFare
  o Activity[] activities
}
```

`FundCampaign` Once the Campaign is completely defined, i.e., all the missions have been added, it can start being funded by the donors. This transaction allows the Donor `donor` to choose a given amount of Educoin (`educoinAmount`) and donate them to the Campaign `campaign`.

```
/**
A donor can give Educoin to a campaign (funding)
*/
transaction FundCampaign{
  o Integer educoinAmount
  --> Campaign campaign
  --> Donor donor
}
```

EnrollStudentToMission When a Campaign is completely funded, students can start enrolling its missions. This transaction allows the current participant (that the access control list forces to be a Student) to engage with the Mission `mission`. If the transaction completes successfully a new mission contract between Student and Mission will be created.

```
/**
A student enroll to an Activity
 */
transaction EnrollStudentToMission {
  --> Mission mission
}
```

StudentAskForMentor After enrolling a mission, the student can ask for a review from the chosen Mentor (`mentor`). For the transaction to complete successfully at least one document must be submitted. The mission contract between Student and Mission will be updated accordingly.

```
transaction StudentAskForMentor {
 --> MissionContract missionContract
  o String[] attachments
  o String[] filenames
 --> Mentor mentor
}
```

MentorReview If a mentor has been chosen by a student to review his documentation, he cannot decline to make a review. A review is said to be done when the mentor has submitted at least one document to the system. The mission contract between Student, Campaign and Mentor will be updated accordingly.

```
transaction MentorReview {
  --> MissionContract missionContract
  o String[] attachments
  o String[] filenames
}
```

StudentLeaveReview Once the student benefit of the mentor services, she can decide to leave a review to the mentor, sharing its experience with the whole platform. A student can only review the mentor that reviews his documentation. The transaction requires the mission contract between Student, Campaign and Mentor in order to select the right mentor, and the content of the review.
Reviews are anonymous.

```
transaction StudentLeaveReview {
  --> MissionContract missionContract
  o Review review
}
```

StudentSubmitMission When the student feels ready, she can submit the produced documentation to the campaign. The provided files (`attachments`) will be added the right contract between Student and Mission. If the Campaign has already evaluated his work, the student cannot submit documentation anymore.

```
transaction StudentSubmitMission {
 --> MissionContract missionContract
  o String[] attachments
  o String[] filenames
}
```

StudentCompleteMission Once the student submitted his documentation the Campaign can start the evaluation. For each activity of the mission, the Campaign can decide whether the documentation submitted by the student completes it or not. The Campaign must provide the transaction with an array of boolean whose indices correspond to the indices of the array of activities (`activities`) declared in the mission. This means `completedActivities[0] = true` represents that the activities contained in `activities[0]` has been successfully completed.

```
transaction StudentCompleteMission {
  -->MissionContract missionContract
  o Boolean[] completedActivities
}
```

AddItem Once the vendor agrees the price with a platform administrator he can add an item to the market place. This part currently presents a few limitations, indeed, it is out of the scope of this PoC defining how exactly the platform administrator and the vendor agree the items (and relative prices) to be added to the market place.

```
/*
* A vendor can create a new item
*/
transaction AddItem {
  o String name
  o String description
  o String cost
}
```

BuyItem The only types of participants that are allowed to buy from the market place are Student and Mentor.

```
transaction BuyItem {
  --> Item item
}
```

The complete `logic.js` file can be found in Appendix A.2.

3.1.4 Access Control List

Hyperledger Composer allows to define a set of access rules inside an Access Control List file (`.acl`). Every rule contains the following fields:

Description: a string describing the goal of the rule.

Participant: the participant class that is the subject of the rule.

Operation: the type of operation (CREATE, READ, UPDATE, DELETE, all)

Resource: the resource class to which the operation refers.

Condition: a simple condition under which the rule counts.

Action: the final decision of the rule (ALLOW, DENY).

As an example, the rule `student R1` states that the participant of type Student can READ the other participants of type Student if and only if their identifier is the same. This basically means that a Student can read only its own record. It perfectly makes sense since for privacy reasons we don't want a student having access to the all the other students' records.

```
rule student_R1 {
  description: "Students can read their record only"
  participant(p): "org.bfore.Student"
  operation: READ
  resource(r): "org.bfore.Student"
  condition: (p.getIdentifier() == r.getIdentifier())
  action: ALLOW
}
```

Instead of showing all the rules (more than 60), for the sake of brevity I prefer to summarize what every participant can do.
Campaigns can:

- Read all the campaigns;
- Create their own missions but only in the context of the `AddMissionToCampaign` transaction;
- Read all the mission contracts related to one of their missions;
- Create the `AddMissionToCampaign` transaction;
- Update its own record but only in the context of a `AddMissionToCampaign` transaction. Indeed the funding goal is computed and updated every time a new mission is added;
- Create the `StudentCompleteMission` transaction;
- Read and update students' records but only in the context of a `StudentCompleteMission` transaction. Indeed, the chaincode needs to update student balance to add the earned Educoin;
- Update their own record but only in the context of a `StudentCompleteMission` transaction. Indeed the Campaign must also upload its own Educoin balance;

- Read all the missions;
- Update the mission contracts but only inside the `StudentCompleteMission` transaction. Indeed, the mission contract must be recorded as completed, and the total amount of Educoin earned by the student must be saved;
- Read all the items;

Students can:

- Read their own record;
- Read their own mission contracts;
- Read all the missions;
- Create the transaction `EnrollStudentToMission`;
- Read all the campaigns;
- Read and update the records of the Vendor but only in the context of a `BuyItem` transaction. This makes sure that the chaincode can correctly update the Educoin balance of the student and of the vendor;
- Update its own record but only in the context of a `BuyItem` transaction, so that the chaincode can correctly update the student balance.
- Read all the items;
- Create the `BuyItem` transaction;
- Create the `StudentSubmitMission` transaction;
- Create a `MissionContract` but only in the context of a `EnrollStudentToMission` transaction;
- Update their own `MissionContract` but only in the context of a `StudentSubmitMission` transaction;
- Update a `mission` but only in the context of a `EnrollStudentToMission` transaction. This is necessary because the current number of students (`currentStudents`) must be updated;
- Update an `Item` but only in the context of a `BuyItem` transaction. This is needed because the chaincode needs to modify the `owner`;
- Create the `StudentAskForMentor` transaction;
- Create the `StudentLeaveReview` transaction;
- Update a mentor record but only in the context of a `StudentLeaveReview` transaction. This is needed because the review must be added to the mentor record;
- Read all the mentors;
- Update a mission contract record but only in the context of a `StudentAskForMentor` transaction. Indeed, the mentor object must be added to the contract.

Mentors can:

- Read all the mentors;
- Create the `MentorReview` transaction;
- Read and upadte a mission contract but only in the context of a `MentorReview` transaction. Indeed the chaincode needs to add the new attachments to the mission contract;
- Read all the missions;
- Read all the campaigns;
- Read all the items;
- Read all the mission contract;
- Update a Campaign but only in the context of a `MentorReview` transaction. Indeed, the chaincode needs to modify the Educoin balance of the campaign;
- Update his own record but only in the context of a `MentorReview` transaction;
- Create the `BuyItem` transaction;
- Read and update a Vendor but only in the context of a `BuyItem` transaction;
- Update his own record but only in the context of a `BuyItem` transaction;
- Update an item but only in the context of a `BuyItem` transaction.

Donors can:

- Read their own record;
- Read all the campaigns;
- Read all the missions;
- Read a `FundCampaign` transaction but only if they are the donor who created it;
- Update their own record but only in the context of a `FundCampaign` transaction;
- Update a campaign record but only in the context of a `FundCampaign` transaction;
- Read all the items. A donor can read them, but cannot buy a item.
- Read all the mission contracts. This is needed to grant transparency. A donor can see how the campaign he funded is spending the money.

Vendors can:

- Read their own record;
- Create the `AddItem` transaction;
- Create an item for which they are the owner but only in the context of an `AddItem` transaction;

- Read all the mission;
- Read all the campaigns;
- Read all the items.

The complete `permissions.acl` file can be found in Appendix A.3.

3.1.5 Queries

Hyperldeger Composer also allows to define simple queries inside a `.qry` file. This feature adds more flexibility. Indeed, instead of just reading the entire resource registry, it makes possible to define some filters.

An Hyperldger Composer query is made of:

`description`: a string describing the query goal;

`statement`: a SQL-like statement defining the content of the query.

As an example, the query Q1, given a student and a mission return the mission contract (if any).

```
query Q1 {
  description: "Given student and mission return the Mission Contract (if
      any)"
  statement:
    SELECT org.bfore.MissionContract
      WHERE(_$mission == mission AND _$student == student)
}
```

In this PoC 9 different queries have been defined. They allow to:

1. Return the mission contract (if any) between a given student and a given mission;
2. Given a `missionId` check if there is already a winner, i.e, if a student already completed all the activities of that mission;
3. Return all the completed campaigns, where completed means that all the missions have been defined and that the campaign's funding goal is well-defined;
4. Given a student return all his mission contracts;
5. Given a participant, return all his items (if any);
6. Given a donor, return all the `FundCampaign` transactions he has done;
7. Get all the missions of a given campaign;
8. Get all the mission contracts of a given missions;
9. Given a mentor, return all his mission contracts.

The complete `queries.qry` file can be found in Appendix A.4.

3.2 Business Network Deployment

3.2.1 Starting Hyperledger Fabric

Hyperledger Fabric can be easily installed following these commands:

```
mkdir ~/fabric-dev-servers
cd ~/fabric-dev-servers
curl -O
    https://raw.githubusercontent.com/hyperledger/composer-tools/master\
    /packages/fabric-dev-servers/fabric-dev-servers.tar.gz
tar -xvf fabric-dev-servers.tar.gz
export FABRIC_VERSION=hlfv11
./downloadFabric.sh
./startFabric.sh
./createPeerAdminCard.sh
```

Once installed, Hyperledger Fabric can be started running the script `startFabric.sh`. The script will leverages different Docker images to quickly bootstrap a Hyperledger Fabric network comprised of one Orderer organization `OrdererOrg` and one Peer oragnization `Org1`. The default orderer implementation is solo. Two default orderer implementations can be used: solo and kafka. However solo must never be used in production.

Four different docker containers will be started:

`ordered.example.com` - Running the image `hyperledger/fabric-orderer`

`ca.org1.example.com` - Running the image `hyperledger/fabric-ca`

`peer0.org1.example.com` - Running the image `hyperledger/fabric-peer`

`couchdb` - Running the image `hyperledger/fabric-couchdb`

All these details can be found in the file `docker-compose.yml`.

For the sake of the PoC this file has not been changed. However, in production this file should the entire network architecture, including at least:

- One organization for each university or school that wants to allow their students and their professors to participate to the B4E platform;
- One organization for each country that wants to run the project, or alternatively just one for entire world.

In Hyperledger Fabric v1.1, a peer can be an administrator or just a member. However only administrator can install Hypereldger Fabric chaincode onto peers so in order to deploy a business network to a set of peers, an identity with administrative rights to all of those peers must be created.

The script `./createPeerAdminCard.sh` creates a Peer Admin business network card using the certificate and private key associated with the peer admin identity for that peer. The peer administrator for the network is called `PeerAdmin` and the identity is automatically imported by the script.

3.3 Business network administrators

Only a *Business Network Archive* (.bna) file can be deployed on top of a Hyperledger Fabric network so the four files (model, logic, query and permissions) must be

packaged into a *Business Network Archive* (.bna) file, using the `composer archive create` command.

Once we obtain the business network archive file we can deploy it to Hyperledger Fabric. To do that the `composer network install` command followed by a `composer network start` command must be used.

```
composer archive create -t dir -n .

composer network install -a b4e-network@0.0.1.bna -c PeerAdmin@hlfv1

composer network start --networkName b4e-network --networkVersion 0.0.1
    --networkAdmin admin --networkAdminEnrollSecret adminpw --card
    PeerAdmin@hlfv1

composer card import --file admin@b4e-network.card
```

In a deployed business network, the access control rules (defined in the .acl file) enforce rights and permissions. In every business network at least one participant, with a valid identity, is required to allow client application to interact with the business network.

In particular, every organization that participates to the business network must have its own business network administrator. This participant type is in charge of configuring the business network for its organization and on-boarding other participants from their organization.

Hyperledger Composer offers a built-in participant type, `org.hyperledger.composer.syster` for this reason.

By default, during deployment a business network administrator participant will be created by Hyperledger Composer, that will also bind to it the identity that has been used for deploying the business network.

Thanks to the additional options of the `composer network start` command, the business network administrators that should be created during the deployment can be specified.

3.3.1 Composer REST Server

Hyperledger Composer includes a standalone Node.js process that exposes a business network as a REST API. This feature is extremely useful, indeed it allows a web application to easily interact with the business network.

Moreover the business network, and so the REST server, need to distinguish between the different types of participants in order to authorize access to resources and allow end users of the blockchain network to interact with the deployed business network. That's why an authentication strategy is needed.
Hyperledger Composer REST server is compatible with a lot of passport strategies, in this PoC the Google+ API has been chosen as the authentication provider as its very easy to setup a Google account.

Google OAUTH2.0 is an "authorization protocol" that can also be used as a "delegated authentication scheme".

As explained in Fig. 3.2 the Composer REST server has been built to provide access to business network resources, while Google+ API Oauth2.0's role is to protect the resources. The resource owner is the Google+ API user account. On requests, the Google+ server requests consent of the resource owner and issues access tokens to REST clients (e.g. web client apps) to allow them to access the protected resources.

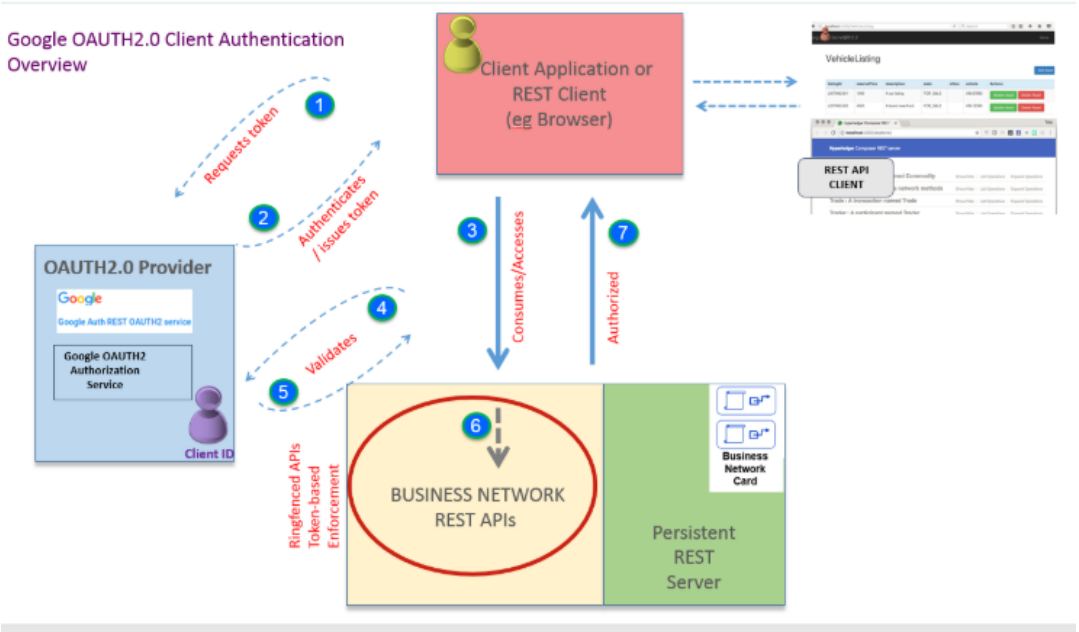


FIGURE 3.2: Google OAUTH2.0 Client Authentication Overview

The access the APIs protected by OAuth2.0 is granted through tokens that are stored in the local storage of the user's web browser, so that they must be requested just once, and the following times the token will be retrieved from the cookie and then validated.

The REST Server itself is configured to persist the business network cards (required to connect to the network) using the MongoDB store.

Here comes an issue. The user is supposed to ask for a business card through the REST API, but at the same time he needs a business card to access the REST API. To solve this loop a second Composer REST Server is needed. The second Composer REST Server will not require authentication, so that the first time an user register to the web application he can ask for a business network card. After obtaining a business network card the user can authenticate to the first REST Server and perform all the operations needed.

However, with an unauthenticated REST server available anybody could potentially perform operations and interact with the business network in a malicious way, so the unauthenticated REST server must have only the rights restricted to the ones needed to create a new user and issue a business network card for him.

The process is explained in Fig. 3.3.

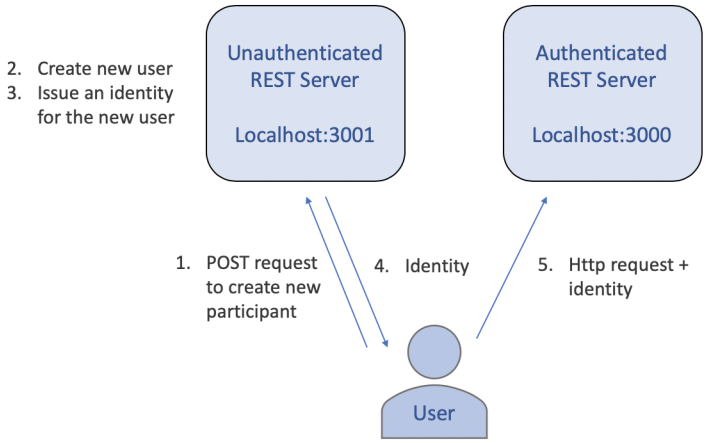


FIGURE 3.3: The first time a new participant wants to register to the platform he interacts with the unauthenticated REST server, which creates a new user and issues a business network card for him. Thanks to business network card, the user can now interact with the authenticated REST server and performs all the operation that card has the rights to.

Since we need two Composer REST servers and we need the unauthenticated one to have limited rights, we need two different cards to associate to the servers. The unauthenticated REST server needs to be able to issue new business network cards, and the only card that has this right is the one we created before `admin@b4e-network` that is bound to the Hyperledger Fabric Peer administrator. This card will be the one with limited rights in all the other operations. For what concerning the authenticated REST Server another card is needed. That is the reason why we will create a new participant:

```
composer participant add -c admin@b4e-network -d
    '{"$class":"org.hyperledger.composer.system.NetworkAdmin",
    "participantId":"restadmin"}'

composer identity issue -c admin@b4e-network -f restadmin.card -u
    restadmin -a
    "resource:org.hyperledger.composer.system.NetworkAdmin#restadmin"

composer card import -f restadmin.card

composer network ping -c restadmin@b4e-network
```

To prevent the unauthenticated REST Server from allowing malicious operations, in the `permissions.acl` file we specify:

```
rule rule1 {
   description: "The NetworkAdmin must be able to create a new
       participant"
   participant: "org.hyperledger.composer.system.NetworkAdmin"
   operation: CREATE, READ
   resource: "org.bfore.Business"
   action: ALLOW
}

rule NetworkAdminUser {
```

```
10    description: "Grant business network administrators full access to
          user resources"
11    participant(p): "org.hyperledger.composer.system.NetworkAdmin"
12    operation: ALL
13    resource(r): "**"
14    condition: (p.getIdentifier() != "admin")
15    action: ALLOW
16 }
```

The READ operation in `rule1` is needed since the participant is read from the participant registry before issuing the card, to be sure that participant really exists.

As mentioned before, the authenticated REST Server is configured to persist the business network cards using the MongoDB store. With the command:

```
1 docker run -d --name mongo --network composer_default -p 27017:27017 mongo
```

an instance of the MongoDB docker container will be started.
We also need to pull the Docker image located at `/hyperledger/composer-rest-server` and additionally install two more npm modules:

- `loopback-connector-mongodb`: this module allows the REST server to use MongoDB as a data source.
- `passport-google-oauth2` - this module allows to authenticate to the REST server using a Google+ account.

```
1 cd $HOME ; mkdir dockertmp
2
3 cd dockertmp
4
5 FROM hyperledger/composer-rest-server
6 RUN npm install --production loopback-connector-mongodb
      passport-google-oauth2 && \
7 npm cache clean --force && \
8 ln -s node_modules .node_modules > Dockerfile
9
10 docker build -t myorg/composer-rest-server .
```

The parameter given the -t flags is the name for this Docker image.
The Composer REST Server needs to be configured through some environment variables. Having a Google+ account allows to create an OAUTH2.0 authentication service for authenticating client application. At the end of the configuration Google provides a clientID and a client secret. With this information it is now possible to create a file called `envars.txt` and configure the authenticated Composer REST Server.

```
1 COMPOSER_CARD=restadmin@b4e-network
2 COMPOSER_NAMESPACES=never
3 COMPOSER_AUTHENTICATION=true
4 COMPOSER_MULTIUSER=true
5 COMPOSER_PROVIDERS='{
6    "google": {
7         "provider": "google",
8         "module": "passport-google-oauth2",
```

```
        "clientID":
            "4626139684-jc70cj9f8nl49n6jtfresqkmst2nfouv.apps.googleusercontent.com",
        "clientSecret": "5IQKO88YZGiwUeuXFJUu4sYB",
        "authPath": "/auth/google",
        "callbackURL": "/auth/google/callback",
        "scope": "https://www.googleapis.com/auth/plus.login",
        "successRedirect":
            "http://localhost:4201/register?authenticated=true",
        "failureRedirect": "/"

    }
}'
COMPOSER_DATASOURCES='{
   "db": {
      "name": "db",
      "connector": "mongodb",
      "host": "mongo"
   }
}'

source envars.txt
```

The environment variables defined will indicate that we are willing to start a multi-user server with authentication using Google OAuth2 along with MongoDB as the persistent data source.

Last but not least, given that we are using Docker containers, 'localhost' address must be changed with docker hostnames and create a new `connection.json` which goes into the card of the REST administrator (restadmin).

```
sed -e 's/localhost:7051/peer0.org1.example.com:7051/' -e
    's/localhost:7053/peer0.org1.example.com:7053/' -e
    's/localhost:7054/ca.org1.example.com:7054/' -e
    's/localhost:7050/orderer.example.com:7050/' <
    $HOME/.composer/cards/restadmin@b4e-network/connection.json >
    /tmp/connection.json && cp -p /tmp/connection.json
    $HOME/.composer/cards/restadmin@b4e-network/
```

It is now possible to run the REST server instance:

```
docker run \
-d \
-e COMPOSER_CARD=${COMPOSER_CARD} \
-e COMPOSER_NAMESPACES=${COMPOSER_NAMESPACES} \
-e COMPOSER_AUTHENTICATION=${COMPOSER_AUTHENTICATION} \
-e COMPOSER_MULTIUSER=${COMPOSER_MULTIUSER} \
-e COMPOSER_PROVIDERS="${COMPOSER_PROVIDERS}" \
-e COMPOSER_DATASOURCES="${COMPOSER_DATASOURCES}" \
-v ~/.composer:/home/composer/.composer \
--name rest \
--network composer_default \
-p 3000:3000 \
myorg/composer-rest-server
```

This REST server instance will be listening at `localhost:3000`.
We now need to run the second, unauthenticated REST server. To do this, we can create a file called `envars2.txt` and configure the unauthenticated REST server.

```
COMPOSER_CARD=admin@b4e-network
COMPOSER_NAMESPACES=never
COMPOSER_AUTHENTICATION=false
COMPOSER_MULTIUSER=false
```

and finally

```
source envars2.txt
```

We will now run a Docker container with this REST server instance in the port 3001.

```
sed -e 's/localhost:7051/peer0.org1.example.com:7051/' -e
    's/localhost:7053/peer0.org1.example.com:7053/' -e
    's/localhost:7054/ca.org1.example.com:7054/' -e
    's/localhost:7050/orderer.example.com:7050/' <
    $HOME/.composer/cards/admin@b4e-network/connection.json >
    /tmp/connection.json && cp -p /tmp/connection.json
    $HOME/.composer/cards/admin@b4e-network/

docker run -d -e COMPOSER_CARD=${COMPOSER_CARD} -e
    COMPOSER_NAMESPACES=${COMPOSER_NAMESPACES} -e
    COMPOSER_AUTHENTICATION=${COMPOSER_AUTHENTICATION} -e
    COMPOSER_MULTIUSER=${COMPOSER_MULTIUSER} -e
    COMPOSER_PROVIDERS="${COMPOSER_PROVIDERS}" -e
    COMPOSER_DATASOURCES="${COMPOSER_DATASOURCES}" -v
    ~/.composer:/home/composer/.composer --name rest2 --network
    composer_default -p 3001:3000 myorg/composer-rest-server
```

This REST server instance will now be listening at `localhost:3001`.

Chapter 4

Conclusions and Future Works

This Master Thesis provides a study about how can Blockchain impact transparency and scalability of Evoke, a World Bank project aimed to inspire youth around the world to develop a passionate curiosity for learning and to produce a Proof of Concept that shows how Evoke architecture can be re-designed using blockchains.

The chosen blockchain framework, Hyperledger Fabric, is an open source enterprise-grade permissioned distributed ledger technology (DLT) platform, established under the Linux Foundation.

Aims of this Master Thesis has been to answer two main research questions:

> *Research Question 1*: How can B4E ensure donated funds are properly maintained and manages (i.e. 100% of designated funds flow to educational purposes)?
>
> *Research Question 2*: How can B4E scale in as many countries as possible minimizing the duplication effort?

This Master Thesis proved how a blockchain framework such as Hyperledger Fabric can grant a complete visibility to donors who want to fund campaigns inside the B4E platform. Donors can track the missions as well as the activities that each campaign created, together with all the contracts made between campaigns, students and mentors.

Blockchain solves one of the main challenge of Evoke: the time taken from when donor funds are received when the donation reports can be sent back to the donors used to range anywhere between a few months to a year, thus discouraging donations. Now, not only donors can have a donation report in real time, but also they can have a complete visibility on the activities created by the campaigns.

Following the network architecture explained in Section 3.2, every new university that wants to allow its students and professors to participate to the B4E network just have to ask the network administrator to add a server (property of the university) as a new peer representing that university.

Currently, Evoke is a standalone, in-country effort led by a not for profit agency in partnership with an academic institution. This basically means that different countries run different versions of Evoke and every time a game is to be launched in a country, the complete system set-up is repeated creating an obstacle and persistent challenges. However, following the new architecture, Evoke can be a world project with campaigns from all over the world. Every new university that wants to allow its students and professors to join the B4E network, just have to buy a server and ask the network administrator to add the server to the network.

The new architecture not only minimize the duplication effort, but also provides to students and professors an international environment, thus increasing competition and motivation. A link to the PoC video demo can be found here.

Appendix A

Business Network Definition Code

A.1 model.cto

```
namespace org.bfore

asset MissionContract identified by missionContractId {
  o String missionContractId
  o Boolean completed default=false
  o Boolean mentorUsed default=false
  o DateTime dueDate
  o DateTime reviewDate optional
  o DateTime mentorSubmissionDate optional
  o DateTime submissionDate optional
  o Integer earnedEducoin default=0
  o Boolean winner default=false
  o Boolean[] completedActivities optional
  o String[] filenames optional
  o String[] attachments optional
  --> Mission mission
  --> Student student
  --> Mentor mentor optional
}

/**
* The activities to be completed inside a mission
 */
concept Activity{
  o String name
  o String description
  o Integer educoin default = 0
}

concept Review {
  o String title
  o String description
  o Double score
}

/**
* A mission to be complete from students to earn EduCoin
*/
asset Mission identified by missionId {
  o String missionId
  o String missionName
  o DateTime dueDate
  o String missionDescription optional
  o Integer educoin default = 0 // = to the sum of the educoin for each
      assignment
  o Integer bonusEducoin
```

```
  o Integer maxStudents
  o Integer mentorFare
  o Integer currentStudents default=0
  o Activity[] activities
  --> Campaign campaign
}

/**
* Items to be sold in the market place
 */
asset Item identified by itemId{
  o String itemId
  o String name
  o String description
  o Integer cost
  --> Business owner
}

/**
* A concept for a simple street address
*/
concept Address {
  o String city
  o String country
  o String street
  o String zip
}

/**
* An abstract participant type in this business network
*/
abstract participant Business identified by email {
  o String email
  o String firstName
  o String lastName
  o Address address optional
  o Integer educoinBalance default=0 optional
}

/**
* A student is a type of participant in the network
*/
participant Student extends Business {
}

/**
* A mentor is a type of participant in the network
*/
participant Mentor extends Business {
  o Double reviewSum default=0.0 optional
  o Double reviewCount default=0.0 optional
  o Review[] reviews optional
}

/**
* A donor is a type of participant in the network
*/
participant Donor extends Business {
}

/**
* A vendor is a type of participant in the network
*/
```

```
participant Vendor extends Business {
}

/**
A campaign is also a participant, since it can transfer and own educoins.
 */
participant Campaign extends Business {
  o String campaignName
  o String campaignDescription
  o Integer fundingGoal default=0
  o Boolean funded // this means that the funding goal has been reached
  o Boolean completed // this means that all the activities have been added
}

/**
A donor can give Educoin to a campaign (funding)
*/
transaction FundCampaign{
  o Integer educoinAmount
  o Integer realAmount optional
  --> Campaign campaign
  --> Donor donor
}

/**
A student enroll to Mission
 */
transaction EnrollStudentToMission {
  --> Mission mission
}

transaction AddMissionToCampaign {
  o String missionName
  o String missionDescription
  o DateTime dueDate
  o Boolean completeCampaign
  o Integer bonusEducoin
  o Integer maxStudents
  o Integer mentorFare
  o Activity[] activities
}

/**
A student complete a mission
 */
transaction StudentCompleteMission {
  -->MissionContract missionContract
  o Boolean[] completedActivities
}

transaction StudentSubmitMission {
 --> MissionContract missionContract
  o String[] attachments
  o String[] filenames
}

transaction StudentAskForMentor {
 --> MissionContract missionContract
  o String[] attachments
  o String[] filenames
 --> Mentor mentor
}

```

```
transaction MentorReview {
  --> MissionContract missionContract
  o String[] attachments
  o String[] filenames
}

transaction StudentLeaveReview {
  --> MissionContract missionContract
  o Review review
}

/**
A student buy an item
 */
transaction BuyItem {
  --> Item item
}

/*
* A vendor can create a new item
*/
transaction AddItem {
  o String name
  o String description
  o String cost
}

/**
Just for initializing a Demo
 */
transaction SetupDemo {
}
```

A.2 logic.js

```
/**
 * Initialize some test assets and participants useful for running a demo.
 * @param {org.bfore.SetupDemo} setupDemo - the SetupDemo transaction
 * @transaction
 */
async function setupDemo(setupDemo){
   const factory = getFactory();
   const NS = 'org.bfore';

   //create a student
   const student = factory.newResource(NS,
       'Student','studentA@email.com');
   student.firstName = 'Student';
   student.lastName = 'A';
   const studentAddress = factory.newConcept(NS,'Address');
   studentAddress.city = 'Washington DC';
   studentAddress.country = 'USA';
   studentAddress.street = '3222 Northampton Street';
   studentAddress.zip = '20015';
   student.address = studentAddress;
   student.educoinBalance = 0;

   const studentRegistry = await getParticipantRegistry(NS + '.Student');
```

```
23    await studentRegistry.addAll([student]) ;
24    var counter = 0;
25    var act_counter = 0;
26    var campaignNames = ['Social Innovation and Development',
          'Entrepreneurship and New Venture'];
27    var campaignDescriptions = ['The goal of the program to help students
          build changemaking skills', 'Entrepreneurship education benefits
          students from all socioeconomic backgrounds because it teaches kids
          to think outside the box'];
28
29   var missionNames = ['Fight World Hunger', 'You Create'];
30   var missionDescriptions = ['For students interested in exploring
          innovative ways to tackle the world s most pressing social
          challenges and improving livelihoods for low-income populations
          domestically and internationally.', 'Think about a product that
          they would love to invent and review some of the basics of the
          invention process.'];
31
32
33  var activityDescriptions = ['Write a research paper with topic: Where
        does our food come from? And more importantly, why does it travel so
        far to reach us?', 'Design a product and develop a business plan'];
34
35    // create 2 campaigns
36    const campaignRegistry = await getParticipantRegistry(NS +
          '.Campaign');
37    for(var i = 0; i<2; i++) {
38       const campaign = factory.newResource(NS,'Campaign', 'campaign' + i
             + '@email.com');
39       campaign.firstName = 'Sara';
40       campaign.lastName = 'Giammusso';
41       campaign.campaignName = campaignNames[i];
42       campaign.campaignDescription = campaignDescriptions[i];
43       campaign.educoinBalance = 0;
44       campaign.fundingGoal = 0;
45
46
47       for (var j=0; j<1; j++) {
48         const missionRegistry = await getAssetRegistry(NS + '.Mission');
49
50         const mission = factory.newResource(NS,'Mission', String(counter));
51
52         mission.missionName = missionNames[counter];
53         mission.missionDescription = missionDescriptions[counter];
54         mission.bonusEducoin = 2;
55         mission.maxStudents = 3;
56         mission.mentorFare = 4;
57         mission.currentStudents = 0;
58         mission.campaign = campaign;
59         mission.dueDate = new Date(2018, 12, 31, 12, 0, 0, 0);
60         counter++;
61         var totalEducoin = 0;
62         mission.activities = [];
63         for(k=0; k<1; k++){
64            var activity = factory.newConcept(NS,'Activity');
65            activity.name = "Activity " + k;
66            activity.description = activityDescriptions[act_counter++];
67            activity.educoin = 3;
68            mission.activities[k] = activity;
69            totalEducoin += activity['educoin'];
70         }
71         mission.educoin = totalEducoin;
72         await missionRegistry.add(mission);
```

```
    }
    campaign.fundingGoal = 23;
    campaign.completed = true;
    campaign.funded = false;
    await campaignRegistry.addAll([campaign]) ;
}



// create a donor
const donor = factory.newResource(NS,'Donor', 'donor@email.com');
donor.firstName ='Donor';
donor.lastName = 'A';
const donorAddress = factory.newConcept(NS,'Address');
donorAddress.city = 'Washington DC';
donorAddress.country = 'USA';
donorAddress.street = '1234 Donor Street';
donorAddress.zip = '20015';
donor.address = donorAddress;
donor.educoinBalance = 100;

const donorRegistry = await getParticipantRegistry(NS + '.Donor');
await donorRegistry.addAll([donor]) ;

// create 2 mentors
const mentor = factory.newResource(NS, 'Mentor','mentor1@email.com');
mentor.firstName = 'Prof. Joanie';
mentor.lastName ='Bedore';
mentor.reviews = [];
const review1 = factory.newConcept(NS,'Review');
review1.title = 'Awesome';
review1.description = 'Dr. Bedore hands down is the best mentor I have
    ever had, she is very kind hearted and pure spirited. Very
    attentive, caring and understanding';
review1.score = 5;
const review2 = factory.newConcept(NS,'Review');
review2.title = 'Nice mentor';
review2.description = 'Amazing. She is a great mentor and helps you no
    matter what the situation.';
review2.score = 4;
mentor.reviews[0] = review1;
mentor.reviews[1] = review2;
mentor.reviewCount = 2;
mentor.reviewSum = 9;
mentor.educoinBalance = 0;

const mentorRegistry = await getParticipantRegistry(NS + '.Mentor');
await mentorRegistry.addAll([mentor]) ;

const mentor2 = factory.newResource(NS, 'Mentor','mentor2@email.com');
mentor2.firstName = 'Prof. Andrew';
mentor2.lastName ='Finn';
mentor2.reviews = [];
const review3 = factory.newConcept(NS,'Review');
review3.title = 'Not that good';
review3.description = 'He is very disorganized, rude and he does not
    really care about the students. ';
review3.score = 2;
const review4 = factory.newConcept(NS,'Review');
review4.title = 'Average';
review4.description = 'Dr. Finn is a good mentor. However, his
    feedbacks are not that detailed. ';
review4.score = 3;
```

```
131    mentor2.reviews[0] = review3;
132    mentor2.reviews[1] = review4;
133    mentor2.reviewCount = 2;
134    mentor2.reviewSum = 5;
135    mentor2.educoinBalance = 0;
136
137
138    await mentorRegistry.addAll([mentor2]) ;
139
140
141    // create a vendor
142    const vendor = factory.newResource(NS, 'Vendor','vendor@email.com');
143    vendor.firstName = 'Vendor';
144    vendor.lastName ='1';
145    const vendorAddress = factory.newConcept(NS,'Address');
146    vendorAddress.city = 'Washington DC';
147    vendorAddress.country = 'USA';
148    vendorAddress.street = '1234 Vendor Street';
149    vendorAddress.zip = '20015';
150    vendor.address = vendorAddress;
151    vendor.educoinBalance = 0;
152
153    const vendorRegistry = await getParticipantRegistry(NS + '.Vendor');
154    await vendorRegistry.addAll([vendor]) ;
155
156    var itemNames = ['The Official Cambridge Guide to IELTS', 'TOPS
           1-Subject Notebooks', 'Arteza Highlighters Set of 60, Bulk Pack of
           Colored Markers'];
157    var itemDescriptions = ['Perfect for students at band 4.0 and above,
           this study guide has EVERYTHING you need to prepare for IELTS',
           'Spiral, 8" x 10-1/2", College Rule, Color Assortment May Vary, 70
           Sheets, 6 Pack', 'Wide and Narrow Chisel Tips, 6 Assorted Neon
           Colors, for Adults & Kids'];
158
159    // create an item
160    for(var i=0; i<3; i++) {
161      const item = factory.newResource(NS, 'Item', String(i));
162      item.owner = vendor;
163      item.name = itemNames[i];
164      item.description = itemDescriptions[i];
165      item.cost = 3;
166
167      const itemRegistry = await getAssetRegistry(NS + '.Item');
168      await itemRegistry.addAll([item]);
169    }
170
171 }
172
173
174 /**
175  * When a student wants to buy an item from the market place
176  * @param {org.bfore.BuyItem} BuyItem - the BuyItem transaction
177  * @transaction
178  */
179 async function BuyItem(buyItem){
180    const NS = 'org.bfore';
181    var currentParticipant = getCurrentParticipant();
182    if(buyItem.item.owner.getType() != "Vendor")
183       throw new Error('Owner is not a vendor');
184    // 1st check: can the student afford it?
185    if (currentParticipant.educoinBalance >= buyItem.item.cost){
186       // the vendor get the money
187       buyItem.item.owner.educoinBalance += buyItem.item.cost;
```

```
188       const vendorRegistry = await getParticipantRegistry(NS + '.Vendor');
189       await vendorRegistry.update(buyItem.item.owner);
190       // the owner changes
191       buyItem.item.owner = currentParticipant;
192       const itemRegistry = await getAssetRegistry(NS + '.Item');
193       await itemRegistry.update(buyItem.item);
194       // the student spent the money
195       currentParticipant.educoinBalance -= buyItem.item.cost;
196       const studentRegistry = await getParticipantRegistry(NS +
              '.Student');
197       await studentRegistry.update(currentParticipant);
198     }
199   else
200     throw new Error('Not enough money');
201 }
202
203 /**
204  * When a mentor review student documents
205  * @param {org.bfore.MentorReview} MentorReview - the MentorReview
        transaction
206  * @transaction
207  */
208 async function MentorReview(mentorReview){
209   const NS = 'org.bfore';
210   if(mentorReview.missionContract.mentorUsed == true)
211     throw new Error('Mentor already used');
212
213   var len = mentorReview.missionContract.attachments.length;
214   for(var i = 0 ; i < mentorReview.attachments.length ; i++ ){
215     mentorReview.missionContract.attachments[len] =
            mentorReview.attachments[i];
216     mentorReview.missionContract.filenames[len] =
            mentorReview.filenames[i];
217     len++;
218    }
219   var today = new Date();
220   mentorReview.missionContract.mentorUsed = true;
221   const missionContractRegistry = await getAssetRegistry(NS +
        '.MissionContract');
222   // Update mission Contract registry
223   await missionContractRegistry.update(mentorReview.missionContract);
224   // Update mentor balance
225   mentorReview.missionContract.mentor.educoinBalance +=
        mentorReview.missionContract.mission.mentorFare;
226   mentorReview.missionContract.mission.campaign.educoinBalance -=
        mentorReview.missionContract.mission.mentorFare;
227   const mentorRegistry = await getParticipantRegistry(NS + '.Mentor');
228   await mentorRegistry.update(mentorReview.missionContract.mentor);
229   // Update campaign balance
230   const campaignRegistry = await getParticipantRegistry(NS + '.Campaign');
231   await
        campaignRegistry.update(mentorReview.missionContract.mission.campaign);
232 }
233
234
235 /**
236  * When a student submit a mission
237  * @param {org.bfore.StudentSubmitMission} StudentSubmitMission - the
        StudentSubmitMission transaction
238  * @transaction
239  */
240 async function StudentSubmitMission(studentSubmitMission){
241   const NS = 'org.bfore';
```

```
242   if(studentSubmitMission.reviewDate == null) {
243     const missionContractRegistry = await getAssetRegistry(NS +
          '.MissionContract');
244     var today = new Date();
245     var len = studentSubmitMission.missionContract.attachments.length ;
246     studentSubmitMission.missionContract.submissionDate = today;
247     for(var i = 0 ; i < studentSubmitMission.attachments.length ; i++ ){
248       studentSubmitMission.missionContract.attachments[len] =
            studentSubmitMission.attachments[i];
249       studentSubmitMission.missionContract.filenames[len] =
            studentSubmitMission.filenames[i];
250       len++;
251     }
252     // Update mission Contract registry
253     await
          missionContractRegistry.update(studentSubmitMission.missionContract);
254   }
255   else
256     throw new Error('Mission already evalued');
257 }
258
259 /**
260  * When a student ask for the mentor to review his attachments
261  * @param {org.bfore.StudentAskForMentor} StudentAskForMentor - the
       StudentAskForMentor transaction
262  * @transaction
263  */
264 async function StudentAskForMentor(studentAskForMentor){
265   const NS = 'org.bfore';
266   if(studentAskForMentor.missionContract.reviewDate == null &&
        studentAskForMentor.missionContract.mentorUsed == false) {
267     const missionContractRegistry = await getAssetRegistry(NS +
          '.MissionContract');
268     var today = new Date();
269     studentAskForMentor.missionContract.attachments = [];
270     studentAskForMentor.missionContract.filenames = [];
271     studentAskForMentor.missionContract.mentorSubmissionDate = today;
272     for(var i = 0 ; i < studentAskForMentor.attachments.length ; i++ ){
273       studentAskForMentor.missionContract.attachments[i] =
            studentAskForMentor.attachments[i];
274       studentAskForMentor.missionContract.filenames[i] =
            studentAskForMentor.filenames[i];
275     }
276     // From this moment this is the mentor for this mission contract
277     studentAskForMentor.missionContract.mentor =
          studentAskForMentor.mentor;
278
279     // Update mission Contract registry
280     await
          missionContractRegistry.update(studentAskForMentor.missionContract);
281   }
282   else
283     throw new Error('Mission already evalued or mentor already used');
284 }
285
286 /**
287  * When the student leaves a review for the mentor
288  * @param {org.bfore.StudentLeaveReview} StudentLeaveReview - the
       StudentLeaveReview transaction
289  * @transaction
290  */
291 async function StudentLeaveReview(studentLeaveReview){
292   const NS = 'org.bfore';
```

```
293   const mentorRegistry = await getParticipantRegistry(NS + '.Mentor');
294   studentLeaveReview.missionContract.mentor.reviews.push(studentLeaveReview.review);
295   studentLeaveReview.missionContract.mentor.reviewSum +=
          studentLeaveReview.review.score;
296   studentLeaveReview.missionContract.mentor.reviewCount++;
297   await mentorRegistry.update(studentLeaveReview.missionContract.mentor);
298 }
299
300
301
302 /**
303  * When a student complete a mission
304  * @param {org.bfore.StudentCompleteMission} StudentCompleteMission - the
        StudentCompleteMission transaction
305  * @transaction
306  */
307 async function StudentCompleteMission(studentCompleteMission){
308
309    const NS = 'org.bfore';
310    if(studentCompleteMission.missionContract.completed == true)
311     throw new Error('Mission already completed');
312
313    /* The goal here is to compute the final earnedEducoin based on the
          contract and on the result of the mission
314    and to assign this amount to the student */
315    var earnedEducoin = 0;
316
317
318    if(studentCompleteMission.completedActivities.length !=
          studentCompleteMission.missionContract.mission.activities.length)
319       throw new Error('Not enough activities specified');
320
321    // 1st check: the activity must have been completed in time
322    if(studentCompleteMission.missionContract.dueDate.getTime()
323       >= studentCompleteMission.missionContract.submissionDate.getTime()){
324
325       // 2nd check: is s/he the winner?
326       // 2a: are there already other winners?
327       let result = await query('Q2',
328                                {
329                                   mission:
                                        `resource:${NS}.Mission#${studentCompleteMission.missionC
330                                });
331
332       // 2b: did s/he completed all the activities?
333       var allDone = true;
334       var bonusEducoin = 0;
335       // Now let's also count the earnedEducoin based on the completed
             activities
336       for(i=0; i< studentCompleteMission.completedActivities.length; i++){
337          studentCompleteMission.missionContract.completedActivities[i] =
                studentCompleteMission.completedActivities[i];
338          if
                (!studentCompleteMission.missionContract.completedActivities[i])
339             allDone = false;
340          else
341             earnedEducoin +=
                   studentCompleteMission.missionContract.mission.activities[i].educoin;
342       }
343
344       if (result.length == 0 && allDone == true){
345          studentCompleteMission.missionContract.winner = true;
```

```
346          bonusEducoin =
                 studentCompleteMission.missionContract.mission.bonusEducoin;
347      }
348
349      studentCompleteMission.missionContract.earnedEducoin =
             earnedEducoin + bonusEducoin;
350      // so that students cannot complete this mission twice
351      studentCompleteMission.missionContract.completed = true;
352      var today = new Date();
353      studentCompleteMission.missionContract.reviewDate = today;
354      // update MissionContract registry
355      const missionContractRegistry = await getAssetRegistry(NS +
             '.MissionContract');
356      await
             missionContractRegistry.update(studentCompleteMission.missionContract);
357
358      // update Student registry
359      studentCompleteMission.missionContract.student.educoinBalance +=
             earnedEducoin;
360      studentCompleteMission.missionContract.student.educoinBalance +=
             bonusEducoin;
361      const studentRegistry = await getParticipantRegistry(NS +
             '.Student');
362      await
             studentRegistry.update(studentCompleteMission.missionContract.student);
363
364      // update Campaign balance
365      studentCompleteMission.missionContract.mission.campaign.educoinBalance
             -= earnedEducoin;
366      studentCompleteMission.missionContract.mission.campaign.educoinBalance
             -= bonusEducoin;
367      const campaignRegistry = await getParticipantRegistry(NS +
             '.Campaign');
368      await
             campaignRegistry.update(studentCompleteMission.missionContract.mission.campaign);
369    }
370   else
371     throw new Error('Due date expired');
372 }
373
374
375 /**
376  * Allows to enroll a student to a mission
377  * @param {org.bfore.EnrollStudentToMission} enrollStudentToMission - the
       EnrollStudentToMission transaction
378  * @transaction
379  */
380 async function EnrollStudentToMission(enrollStudentToMission){
381    const NS = 'org.bfore';
382    const factory = getFactory();
383    var currentParticipant = getCurrentParticipant();
384    // 1st check: Students can start only the missions whose campaigns
385    // are already completed and funded
386    // Also there must be still availability for that mission
387    if(enrollStudentToMission.mission.campaign.funded == true &&
388          enrollStudentToMission.mission.currentStudents <
                 enrollStudentToMission.mission.maxStudents){
389       // 2nd check: Students cannot enroll the same mission twice
390       let result = await query('Q1',
391                                {student:
                                     `resource:${NS}.Student#${currentParticipant.email}`,
392                                 mission:
                                      `resource:${NS}.Mission#${enrollStudentToMission.mission.missionId
```

```
393       });
394
395       if (result.length == 0){
396          const missionContractRegistry = await getAssetRegistry(NS +
               '.MissionContract');
397          let existingAssets = await missionContractRegistry.getAll();
398
399          let numberOfAssets = 0;
400          await existingAssets .forEach(function (asset) {
401             numberOfAssets ++;
402          });
403
404          let missionContractId = String(numberOfAssets +1);
405          const missionContract =
                 factory.newResource(NS,'MissionContract', missionContractId);
406          missionContract.dueDate = enrollStudentToMission.mission.dueDate;
407          missionContract.mission = enrollStudentToMission.mission;
408          missionContract.student = currentParticipant;
409          missionContract.completedActivities = [];
410          missionContract.completed = false;
411          // update MissionContract registry
412
413          await missionContractRegistry.add(missionContract);
414
415          enrollStudentToMission.mission.currentStudents += 1;
416          // update Mission registry
417          const missionRegistry = await getAssetRegistry(NS + '.Mission');
418          await missionRegistry.update(enrollStudentToMission.mission);
419       }
420       else throw new Error('Student cannot enroll the same mission
             twice');
421    }
422   else throw new Error('Campaign not available');
423
424 }
425
426
427
428 /**
429  * Allows to a create a mission and add it to a campaign
430  * @param {org.bfore.AddMissionToCampaign} addMissionToCampaign - the
       AddMissionToCampaign transaction
431  * @transaction
432  */
433 async function addMissionToCampaign(addMissionToCampaign){
434    const NS = 'org.bfore';
435    var currentParticipant = getCurrentParticipant();
436    // A mission can be added only if the campaign is not already completed
437    if (currentParticipant.completed == false){
438     // MissionID creation
439     const missionRegistry = await getAssetRegistry(NS + '.Mission');
440     let existingAssets = await missionRegistry.getAll();
441
442     let numberOfAssets = 0;
443     await existingAssets .forEach(function (asset) {
444      numberOfAssets ++;
445     });
446
447     let missionId = String(numberOfAssets +1);
448      const factory = getFactory();
449      const mission = factory.newResource(NS,'Mission', missionId);
450      mission.missionName = addMissionToCampaign.missionName;
```

```
451       mission.missionDescription =
              addMissionToCampaign.missionDescription;
452       mission.bonusEducoin = addMissionToCampaign.bonusEducoin;
453       mission.maxStudents = addMissionToCampaign.maxStudents;
454       mission.mentorFare = addMissionToCampaign.mentorFare;
455       mission.dueDate = addMissionToCampaign.dueDate;
456       mission.currentStudents = 0;
457       mission.activities = [];
458       var totalEducoin = 0;
459       for(i=0; i<addMissionToCampaign.activities.length; i++){
460          mission.activities[i] = addMissionToCampaign.activities[i];
461          totalEducoin += addMissionToCampaign.activities[i].educoin;
462       }
463       mission.educoin = totalEducoin;
464
465       //activity.activityType = addActivityToCampaign.activityType;
466
467
468       mission.campaign = currentParticipant;
469       // MAX Money needed = each mission can be done by a maximum number
              of students per mission, and the winner also earn a bonus
470       currentParticipant.fundingGoal += mission.educoin *
              mission.maxStudents;
471       currentParticipant.fundingGoal += mission.bonusEducoin;
472       currentParticipant.fundingGoal += mission.mentorFare *
              mission.maxStudents;
473       // This may be the last mission for that campaign...
474       if (addMissionToCampaign.completeCampaign == true){
475          currentParticipant.completed = true;
476       }
477
478       // update Mission registry
479
480       await missionRegistry.add(mission);
481       // update Campaign registry
482       const campaignRegistry = await getParticipantRegistry(NS +
              '.Campaign');
483       await campaignRegistry.update(currentParticipant);
484    }
485    else throw new Error('Campaign already completed');
486 }
487
488
489
490 /**
491  * Allows to a donor to fund a chosen campaign
492  * @param {org.bfore.FundCampaign} fundCampaign - the FundCampaign
        transaction
493  * @transaction
494  */
495 async function fundCampaign(fundCampaign){
496    const NS = 'org.bfore';
497    var currentParticipant = getCurrentParticipant();
498    // Check if the campaign can be funded
499    // Campaign can be funded if it hasn't reach the funding goal yet and
          if it is complete
500    // i.e. all the missions have been added.
501    if (fundCampaign.campaign.funded == false &&
           fundCampaign.campaign.completed == true &&
           fundCampaign.educoinAmount > 0){
502       if (fundCampaign.educoinAmount > fundCampaign.campaign.fundingGoal
              - fundCampaign.campaign.educoinBalance )
```

```
503         amount = fundCampaign.campaign.fundingGoal -
                fundCampaign.campaign.educoinBalance;
504       else
505         amount = fundCampaign.educoinAmount;
506       fundCampaign.realAmount = amount;
507
508       //Check if the donor has enough money
509       if (currentParticipant.educoinBalance < amount){
510         throw new Error('Not enough money');
511       }
512
513       //Check if now the campaign is totally funded
514       if (fundCampaign.campaign.educoinBalance + amount ==
              fundCampaign.campaign.fundingGoal){
515         fundCampaign.campaign.funded = true;
516       }
517
518
519       fundCampaign.campaign.educoinBalance += amount;
520       currentParticipant.educoinBalance -= amount;
521
522       // update Campaign registry
523       const campaignRegistry = await getParticipantRegistry(NS +
              '.Campaign');
524       await campaignRegistry.update(fundCampaign.campaign);
525       // update Donor
526       const donorRegistry = await getParticipantRegistry(NS + '.Donor');
527       await donorRegistry.update(currentParticipant);
528     }
529   else throw new Error('Error');
530
531 }
532
533 /**
534  * Allows to a vendor to create a new item
535  * @param {org.bfore.AddItem} addItem - the AddItem transaction
536  * @transaction
537  */
538 async function addItem(addItem){
539     const NS = 'org.bfore';
540     var currentParticipant = getCurrentParticipant();
541     const factory = getFactory();
542
543     // ItemID creation
544     const itemRegistry = await getAssetRegistry(NS + '.Item');
545     let existingAssets = await itemRegistry.getAll();
546
547     let numberOfAssets = 0;
548     await existingAssets .forEach(function (asset) {
549      numberOfAssets ++;
550     });
551
552     let itemId = String(numberOfAssets +1);
553
554     const item = factory.newResource(NS,'Item', itemId);
555     item.name = addItem.name;
556     item.description = addItem.description;
557     item.cost = addItem.cost;
558     item.owner = currentParticipant;
559
560     // update Item registry
561
562     await itemRegistry.add(item);
```

```
563
564 }
```

A.3 permissions.acl

```
1  // Students: 20 rules
2  rule student_R1 {
3    description: "Students can read their record only"
4    participant(p): "org.bfore.Student"
5    operation: READ
6    resource(r): "org.bfore.Student"
7    condition: (p.getIdentifier() == r.getIdentifier())
8    action: ALLOW
9  }
10
11 rule student_R2 {
12   description: "Students can read only their MissionContract"
13   participant(p): "org.bfore.Student"
14   operation: READ
15   resource(r): "org.bfore.MissionContract"
16   condition: (p.getIdentifier() == r.student.getIdentifier())
17   action: ALLOW
18 }
19
20 rule student_R3 {
21   description: "Students can read all the missions"
22   participant: "org.bfore.Student"
23   operation: READ
24   resource: "org.bfore.Mission"
25   action: ALLOW
26 }
27
28 rule student_R4{
29   description: "Students can make the transaction enroll to mission"
30   participant: "org.bfore.Student"
31   operation: CREATE
32   resource: "org.bfore.EnrollStudentToMission"
33   action: ALLOW
34 }
35
36 // Also without this Student cannot do the transaction
       EnrollStudentToActivity
37 rule student_R5{
38   description: "Students can read the campaign"
39   participant: "org.bfore.Student"
40   operation: READ
41   resource: "org.bfore.Campaign"
42   action: ALLOW
43 }
44
45 rule student_R6 {
46   description: "Students can read and update vendors balance in the
         BuyItem transaction"
47   participant: "org.bfore.Student"
48   operation: READ, UPDATE
49   resource: "org.bfore.Vendor"
50   transaction: "org.bfore.BuyItem"
51   action: ALLOW
```

```
52 }
53
54 rule student_R7 {
55   description: "Students can read items"
56   participant: "org.bfore.Student"
57   operation: READ
58   resource: "org.bfore.Item"
59   action: ALLOW
60 }
61
62 rule student_R8 {
63   description: "Students can make the BuyItem transaction"
64   participant: "org.bfore.Student"
65   operation: CREATE
66   resource: "org.bfore.BuyItem"
67   action: ALLOW
68 }
69
70
71 rule student_R10 {
72  description: "Student can make a StudentSubmitMission transaction"
73  participant: "org.bfore.Student"
74  operation: CREATE
75  resource: "org.bfore.StudentSubmitMission"
76  action: ALLOW
77 }
78
79 rule student_R11 {
80   description: "Students can update their record only in the context of
       the BuyItem transaction"
81   participant(p): "org.bfore.Student"
82   operation: UPDATE
83   resource(r): "org.bfore.Student"
84   transaction(tx): "org.bfore.BuyItem"
85   condition: (p.getIdentifier() == r.getIdentifier())
86   action: ALLOW
87 }
88
89 rule student_R12 {
90   description: "Students can create a MissionContract in the
       EnrollStudentToMissiontransaction"
91   participant: "org.bfore.Student"
92   operation: CREATE
93   resource: "org.bfore.MissionContract"
94   transaction: "org.bfore.EnrollStudentToMission"
95   action: ALLOW
96 }
97
98 rule student_R13 {
99   description: "Students can update their MissionContract in the
       StudentSubmitMission transaction"
100  participant(p): "org.bfore.Student"
101  operation: UPDATE
102  resource(r): "org.bfore.MissionContract"
103  transaction(tx): "org.bfore.StudentSubmitMission"
104  condition: (p.getIdentifier() == r.student.getIdentifier())
105  action: ALLOW
106 }
107
108 rule student_R14 {
109   description: "Students can update the missions in the
       EnrollStudentToMission transaction, to modify currentStudents"
110   participant: "org.bfore.Student"
```

```
111   operation: UPDATE
112   resource: "org.bfore.Mission"
113   transaction: "org.bfore.EnrollStudentToMission"
114   action: ALLOW
115 }
116
117 rule student_R15{
118   description: "Students can modify items owner in the BuyItem transaction"
119   participant: "org.bfore.Student"
120   operation: UPDATE
121   resource: "org.bfore.Item"
122   transaction: "org.bfore.BuyItem"
123   action: ALLOW
124 }
125
126 rule student_R16 {
127   description: "Students can read all the missions contracts"
128   participant: "org.bfore.Student"
129   operation: READ
130   resource: "org.bfore.MissionContract"
131   action: ALLOW
132 }
133
134 rule student_R17 {
135   description: "Students can make the StudentAskForMentor tx"
136   participant: "org.bfore.Student"
137   operation: CREATE
138   resource: "org.bfore.StudentAskForMentor"
139   action: ALLOW
140 }
141 rule student_R18 {
142   description: "Students can make the StudentLeaveReview tx"
143   participant: "org.bfore.Student"
144   operation: CREATE
145   resource: "org.bfore.StudentLeaveReview"
146   action: ALLOW
147 }
148 rule student_R19 {
149   description: "Students can update the mentor record in the
        StudentLeaveReview tx"
150   participant: "org.bfore.Student"
151   operation: UPDATE
152   resource: "org.bfore.Mentor"
153   transaction: "org.bfore.StudentLeaveReview"
154   action: ALLOW
155 }
156 rule student_R20 {
157   description: "Students can update the mission contract record in the
        StudentAskForMentor tx"
158   participant: "org.bfore.Student"
159   operation: UPDATE
160   resource: "org.bfore.MissionContract"
161   transaction: "org.bfore.StudentAskForMentor"
162   action: ALLOW
163 }
164 rule student_R21 {
165   description: "Student can read all the mentors"
166   participant: "org.bfore.Student"
167   operation: READ
168   resource: "org.bfore.Mentor"
169   action: ALLOW
170 }
171
```

```
172
173 /**
174 * CAMPAIGN'S RULES
175 */
176 // Campaign: 10 Rules
177 rule campaign_R1 {
178   description: "Campaigns can read other campaigns"
179   participant: "org.bfore.Campaign"
180   operation: READ
181   resource: "org.bfore.Campaign"
182   action: ALLOW
183 }
184
185 rule campaign_R2 {
186   description: "Campaigns can create their missions"
187   participant(p): "org.bfore.Campaign"
188   operation: CREATE
189   resource(r): "org.bfore.Mission"
190   transaction: "org.bfore.AddMissionToCampaign"
191   condition: (p.getIdentifier() == r.campaign.getIdentifier())
192   action: ALLOW
193 }
194
195 rule campaign_R3 {
196   description: "Campaigns can read all the mission contracts for their
        missions"
197   participant(p): "org.bfore.Campaign"
198   operation: READ
199   resource(r): "org.bfore.MissionContract"
200   condition: (p.getIdentifier() == r.mission.campaign.getIdentifier())
201   action: ALLOW
202 }
203
204 rule campaign_R4 {
205   description: "Campaign can make the AddMissionToCampaign tx"
206   participant: "org.bfore.Campaign"
207   operation: CREATE
208   resource: "org.bfore.AddMissionToCampaign"
209   action: ALLOW
210 }
211
212 rule campaign_R5 {
213   description: "Campaign can make the StudentompleteMission tx"
214   participant(p): "org.bfore.Campaign"
215   operation: CREATE
216   resource(r): "org.bfore.StudentCompleteMission"
217   condition: (p.getIdentifier() ==
        r.missionContract.mission.campaign.getIdentifier())
218   action: ALLOW
219 }
220
221 rule campaign_R6 {
222   description: "Campaign can read and update student balance in the
        StudentCompleteMission tx"
223   participant: "org.bfore.Campaign"
224   operation: READ, UPDATE
225   resource: "org.bfore.Student"
226   transaction: "org.bfore.StudentCompleteMission"
227   action: ALLOW
228 }
229
230 rule campaign_R7 {
```

```
231   description: "Campaigns can update their record in the
          StudentCompleteMission tx"
232   participant(p): "org.bfore.Campaign"
233   operation: UPDATE
234   resource(r): "org.bfore.Campaign"
235   transaction(tx): "org.bfore.StudentCompleteMission"
236   condition: (p.getIdentifier() == r.getIdentifier())
237   action: ALLOW
238 }
239
240 rule campaign_R8 {
241   description: "Campaigns can read all the missions"
242   participant: "org.bfore.Campaign"
243   operation: READ
244   resource: "org.bfore.Mission"
245   action: ALLOW
246 }
247
248 rule campaign_R9 {
249   description: "Campaigns can update the mission contracts for their
          missions in the StudentCompleteMission tx"
250   participant(p): "org.bfore.Campaign"
251   operation: UPDATE
252   resource(r): "org.bfore.MissionContract"
253   transaction(tx): "org.bfore.StudentCompleteMission"
254   condition: (p.getIdentifier() == r.mission.campaign.getIdentifier())
255   action: ALLOW
256 }
257
258 rule campaign_R10 {
259   description: "Campaign can read all the items"
260   participant: "org.bfore.Campaign"
261   operation: READ
262   resource: "org.bfore.Item"
263   action: ALLOW
264 }
265
266 rule campaign_R11 {
267   description: "Campaign can update themselves in the AddMissionToCampaign
          tx"
268   participant: "org.bfore.Campaign"
269   operation: UPDATE
270   resource: "org.bfore.Campaign"
271   transaction: "org.bfore.AddMissionToCampaign"
272   action: ALLOW
273 }
274
275 /**
276 * DONOR'S RULES
277 */
278 // Donor: 7 rules
279 rule donor_R1 {
280   description: "Donors can read their record only"
281   participant(p): "org.bfore.Donor"
282   operation: READ
283   resource(r): "org.bfore.Donor"
284   condition: (p.getIdentifier() == r.getIdentifier())
285   action: ALLOW
286 }
287
288 rule donor_R2 {
289   description: "Donors can read all the campaigns"
290   participant: "org.bfore.Donor"
```

```
291   operation: READ
292   resource: "org.bfore.Campaign"
293   action: ALLOW
294 }
295
296 rule donor_R3 {
297   description: "Donors can read all the missions"
298   participant: "org.bfore.Donor"
299   operation: READ
300   resource: "org.bfore.Mission"
301   action: ALLOW
302 }
303
304 rule donor_R4 {
305   description: "Donors can make and read the FundCampaign tx"
306   participant: "org.bfore.Donor"
307   operation: CREATE, READ
308   resource: "org.bfore.FundCampaign"
309   action: ALLOW
310 }
311
312 rule donor_R5 {
313   description: "Donors can update their record in the FundCampaign tx"
314   participant(p): "org.bfore.Donor"
315   operation: UPDATE
316   resource(r): "org.bfore.Donor"
317   transaction(tx): "org.bfore.FundCampaign"
318   condition: (p.getIdentifier() == r.getIdentifier())
319   action: ALLOW
320 }
321
322 rule donor_R6 {
323   description: "Donors can update the campaigns in the FundCampaign tx"
324   participant: "org.bfore.Donor"
325   operation: UPDATE
326   resource: "org.bfore.Campaign"
327   transaction: "org.bfore.FundCampaign"
328   action: ALLOW
329 }
330
331 rule donor_R7 {
332   description: "Donors can read all the items"
333   participant: "org.bfore.Donor"
334   operation: READ
335   resource: "org.bfore.Item"
336   action: ALLOW
337 }
338 rule donor_R8 {
339   description: "Donors can read all the mission contracts"
340   participant: "org.bfore.Donor"
341   operation: READ
342   resource: "org.bfore.MissionContract"
343   action: ALLOW
344 }
345
346 /**
347 * MENTOR'S RULES
348 */
349 rule mentor_R1 {
350   description: "Mentor can make the MentorReview tx"
351   participant: "org.bfore.Mentor"
352   operation: CREATE
353   resource: "org.bfore.MentorReview"
```

```
354   action: ALLOW
355 }
356 rule mentor_R2 {
357   description: "Mentor can read and update the MissionContract in the
          MentorReview tx"
358   participant: "org.bfore.Mentor"
359   operation: READ,UPDATE
360   resource: "org.bfore.MissionContract"
361   transaction: "org.bfore.MentorReview"
362   action: ALLOW
363 }
364 rule mentor_R3 {
365   description: "Mentor can read all the missions"
366   participant: "org.bfore.Mentor"
367   operation: READ
368   resource: "org.bfore.Mission"
369   action: ALLOW
370 }
371 rule mentor_R4 {
372   description: "Mentor can read all the campaigns"
373   participant: "org.bfore.Mentor"
374   operation: READ
375   resource: "org.bfore.Campaign"
376   action: ALLOW
377 }
378 rule mentor_R5 {
379   description: "Mentor can read all the items"
380   participant: "org.bfore.Mentor"
381   operation: READ
382   resource: "org.bfore.Item"
383   action: ALLOW
384 }
385 rule mentor_R6 {
386   description: "Mentor can read all the missions contracts"
387   participant: "org.bfore.Mentor"
388   operation: READ
389   resource: "org.bfore.MissionContract"
390   action: ALLOW
391 }
392
393 rule mentor_R8 {
394   description: "Mentor can update Campaign Balance in the MentorReview tx"
395   participant: "org.bfore.Mentor"
396   operation: UPDATE
397   resource: "org.bfore.Campaign"
398   transaction: "org.bfore.MentorReview"
399   action: ALLOW
400 }
401 rule mentor_R9 {
402   description: "Mentor can update its own record in the mentor review tx"
403   participant(p): "org.bfore.Mentor"
404   operation: UPDATE
405   resource(r): "org.bfore.Mentor"
406   transaction: "org.bfore.MentorReview"
407   condition: (p.getIdentifier() == r.getIdentifier())
408   action: ALLOW
409 }
410
411 rule mentor_R10 {
412   description: "Mentor can make the buyItem transaction"
413   participant: "org.bfore.Mentor"
414   operation: CREATE
415   resource: "org.bfore.BuyItem"
```

```
  action: ALLOW
}

rule mentor_R11 {
  description: "Mentor can read and update vendors balance in the BuyItem
      transaction"
  participant: "org.bfore.Mentor"
  operation: READ, UPDATE
  resource: "org.bfore.Vendor"
  transaction: "org.bfore.BuyItem"
  action: ALLOW
}

rule mentor_R12 {
  description: "Mentor can update their record only in the context of the
      BuyItem transaction"
  participant(p): "org.bfore.Mentor"
  operation: UPDATE
  resource(r): "org.bfore.Mentor"
  transaction(tx): "org.bfore.BuyItem"
  condition: (p.getIdentifier() == r.getIdentifier())
  action: ALLOW
}

rule mentor_R14{
  description: "Mentors can modify items owner in the BuyItem transaction"
  participant: "org.bfore.Mentor"
  operation: UPDATE
  resource: "org.bfore.Item"
  transaction: "org.bfore.BuyItem"
  action: ALLOW
}

/**
* VENDOR'S RULES
*/
// Vendor: 5 Rules
rule vendor_R1 {
  description: "Vendor can read their record only"
  participant(p): "org.bfore.Vendor"
  operation: READ
  resource(r): "org.bfore.Vendor"
  condition: (p.getIdentifier() == r.getIdentifier())
  action: ALLOW
}

rule vendor_R2 {
  description: "Vendor can create items for which they are the owners"
  participant(p): "org.bfore.Vendor"
  operation: CREATE
  resource(r): "org.bfore.Item"
  transaction: "org.bfore.AddItem"
  condition: (p.getIdentifier() == r.owner.getIdentifier())
  action: ALLOW
}

rule vendor_R3 {
  description: "Vendor can read all the campaigns"
  participant: "org.bfore.Vendor"
  operation: READ
  resource: "org.bfore.Campaign"
  action: ALLOW
}
```

```
477
478 rule vendor_R4 {
479   description: "Vendor can read all the missions"
480   participant: "org.bfore.Vendor"
481   operation: READ
482   resource: "org.bfore.Mission"
483   action: ALLOW
484 }
485
486 rule vendor_R5 {
487   description: "Vendor can read all the items"
488   participant: "org.bfore.Vendor"
489   operation: READ
490   resource: "org.bfore.Item"
491   action: ALLOW
492 }
493
494 rule SystemACL {
495     description: "System ACL to permit all access"
496     participant: "org.hyperledger.composer.system.Participant"
497     operation: ALL
498     resource: "org.hyperledger.composer.system.**"
499     action: ALLOW
500 }
501
502 rule rule1 {
503     description: "Grant business network administrators full access to
          user resources"
504     participant: "org.hyperledger.composer.system.NetworkAdmin"
505     operation: CREATE, READ
506     resource: "org.bfore.Business"
507     action: ALLOW
508 }
509
510
511 rule NetworkAdminUser {
512     description: "Grant business network administrators full access to
          user resources"
513     participant(p): "org.hyperledger.composer.system.NetworkAdmin"
514     operation: ALL
515     resource(r): "**"
516     condition: (p.getIdentifier() != "admin")
517     action: ALLOW
518 }
519
520 rule NetworkAdminSystem {
521     description: "Grant business network administrators full access to
          system resources"
522     participant: "org.hyperledger.composer.system.NetworkAdmin"
523     operation: ALL
524     resource: "org.hyperledger.composer.system.**"
525     action: ALLOW
526 }
```

A.4 queries.qry

```
1 query Q1 {
2   description: "Given student and mission return the Mission Contract (if
        any)"
```

```
  statement:
    SELECT org.bfore.MissionContract
      WHERE(_$mission == mission AND _$student == student)
}

query Q2 {
  description: "Given a missionID check if there is already a winner"
  statement:
    SELECT org.bfore.MissionContract
      WHERE(_$mission == mission AND winner == true)
}

query Q3 {
  description: "Get completed campaigns"
  statement:
    SELECT org.bfore.Campaign
      WHERE(completed == true)
}


query Q5 {
  description: "Get the missions of a student"
  statement:
    SELECT org.bfore.MissionContract
      WHERE(_$student == student)

}

query Q6 {
  description: "Get items by ownerID"
  statement:
    SELECT org.bfore.Item
      WHERE(_$owner == owner)
}

query Q7 {
  description: "Get FundCampaign transaction by donor"
  statement:
    SELECT org.bfore.FundCampaign
      WHERE(_$donor == donor)
}

query Q8 {
 description: "Get all the missions of a given campaign"
 statement:
  SELECT org.bfore.Mission
    WHERE(_$campaign == campaign)
}

query Q9 {
  description: "Get the mission contrats of a given mission"
  statement:
    SELECT org.bfore.MissionContract
      WHERE(_$mission == mission)

}

query Q10 {
  description: "Get the missions of a mentor"
  statement:
    SELECT org.bfore.MissionContract
      WHERE(_$mentor == mentor)

```

```
66 }
```

Bibliography

[1] "A Blockchain Platform for the Enterprise". In: *Hyperledger Fabric Documentation* (2018). URL: https://hyperledger-fabric.readthedocs.io/en/release-1.3/.

[2] Robert Hawkins Barbara Freeman. "Developing Skills in Youth to Solve the World's Most Complex Problems: The Social Innovators' Framework". In: (2017). URL: https://openknowledge.worldbank.org/bitstream/handle/10986/26106/112721-WP-EVOKESocialInnovatorsFrameworkSABERICTno-PUBLIC.pdf?sequence=1&isAllowed=y.

[3] "Build a blockchain insurance app". In: *IBM developer* (2017). URL: https://developer.ibm.com/patterns/build-a-blockchain-insurance-app/.

[4] "Creating a Trusted Experience with Blockchain". In: *Sony Global Education* (2018). URL: https://blockchain.sonyged.com.

[5] "EVOKE - An online alternate reality game supporting social innovation among young people around the world". In: (2017). URL: http://www.worldbank.org/en/topic/edutech/brief/evoke-an-online-alternate-reality-game-supporting-social-innovation-among-young-people-around-the-world.

[6] Stuart Haber and W. Scott Stornetta. "How to time-stamp a digital document". In: *Journal of Cryptology* 3 (Jan. 1991), pp. 99–111. URL: https://link.springer.com/article/10.1007/BF00196791.

[7] "Hash Functions". In: *Wikipedia* (2018). URL: https://en.wikipedia.org/wiki/Hash_function.

[8] P. Sandner M. Valenta. "Comparison of Ethereum, Hyperledger Fabric and Corda". In: (2017). URL: http://explore-ip.com/2017_Comparison-of-Ethereum-Hyperledger-Corda.pdf.

[9] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2008). URL: https://bitcoin.org/bitcoin.pdf.

[10] "Opet Foundation". In: *Opet Foundation* (2018). URL: https://opetfoundation.com.