ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

BLOCKCHAIN AND BEYOND: PROACTIVE LOGIC SMART CONTRACTS

Tesi in
SISTEMI AUTONOMI

Relatore
Prof. ANDREA OMICINI

Co-relatore
Dott. GIOVANNI CIATTO

Presentata da
ALFREDO MAFFI

Seconda Sessione di Laurea
Anno Accademico 2017 – 2018

Abstract

Blockchain-based smart contracts are computer programs which run on top of a blockchain in order to enforce the terms of an agreement between mutually-untrusted parties without the need of a trusted intermediary. With their actual implementations, smart contracts are *passive* entities, that is, they do nothing until one of the parties explicitly trigger them. As a result, they are not able to "actively" participate in the execution of the agreement. Furthermore, since they are deployed on the blockchain, their source code is immutable and cannot be adapted as changes in the real world occur over time. In this thesis, we rethink blockchain-based smart contracts as *proactive* and *logic* entities to overcome the aforementioned issues. In our vision, smart contracts are "proactive" in the sense that they can act without necessarily being triggered by one of their parties, and "logic" in the sense that their business logic is expressed by means of logic programming, allowing for a controllable mutability of their behaviour over time through meta programming. In this work, we analyse the problems which arise as soon as smart contracts are designed as proactive entities, carrying out a feasibility study for their implementation in the first place. Subsequently, we implement a system which supports their execution as a *proof-of-concept* for our idea. Finally, we show how our proactive smart contracts can be used to further enforce the terms of a contract with respect to three different use cases. This work represents a first step towards our final end, which consists in the realization of fully *autonomous* smart contracts, able to reason about the world and automatically act against violations committed with respect to the agreement's terms.

Keywords: blockchain, proactive smart contracts, distributed ledger, logic programming

The message dies, in envelopes
The words divide
The vanity prospers on solo lectures behind walls
Unite the peace of silence
Among the loud it whispers
"Exist, be quiet, and listen"
— **Destrage**

Acknowledgements

I would like to thank Prof. Andrea Omicini for offering me the opportunity to carry out this work and for his wonderful teachings throughout the last two years. I would like to thank Dott. Giovanni Ciatto for supervising my work and for his patience, his great support, and his trust in me.

I would like to thank my friends and colleagues Manuel and Gabriele, for being wonderful companions throughout this long journey and beyond.

I would like to thanks my parents, Manuela and Massimo, and my aunt Maria, for their everlasting love and for always supporting me and my decisions.

Finally, I would like to thank Savant, Destrage, and wac for their music, which helped me a lot during these years.

Contents

Chapter 1

Introduction

Over the past decade, the scientific community has witnessed the ever-growing attention gained by the disruptive technology which goes by the name of *blockchain*. First introduced in 2008, the blockchain can be conceived as a novel solution to the age-old human problem of trust, within the context of distributed systems. It offers an architecture which allows mutually-untrusted parties to trust the system outputs without the need of a central, trusted entity. From a technical standpoint, a blockchain is a shared, append-only, transparent, and distributed data structure which stores *transactions* in linked blocks, which form a chain. A transaction can be conceived as a change made by a certain system user to some common data. The blockchain is replicated over multiple nodes linked in a peer-to-peer network, and each one of its copies keeps tracks of the same *order* and *timing* of these transactions, which are secured through the use of cryptographic schemas. As a result, malicious system users *cannot* tamper with transactions, that is, modify or remove the ones which have been already inserted into a block. This interesting property allows system users to trust the state of the common data, which is the result of all the transactions included in the blockchain, without trusting each others. New transactions are added consistently to each copy of the blockchain (i.e. in the same block *and* in the same order) through a *consensus* protocol, which nodes employ to agree on the next block which will be appended to each local copy.

Even if the primary use of blockchains is as a distributed ledger for cryptocurrencies, they can be used as a system backbone for the realization of a wide range of applications, such as smart property applications or naming and voting systems. Despite their different purposes, all of these applications rely on a blockchain as a way for securely storing data without a single point-of-trust within the system.

Nowadays, there are several blockchain implementations which support the

creation of *smart contracts* to mediate interaction between untrusted parties. In this context, a smart contract is essentially a computer program which is deployed "on the blockchain" by users, meant to act as an intermediary between them while reducing failures and enforcing trust. Smart contracts are capable of managing assets (e.g. amount of cryptocurrencies) and interact *synchronously* with each others. They can be executed by users by means of special transactions.

The concept of smart contracts was first introduced in 1994 as "computerized transactions protocol that executes the terms of a contract" [5]. While blockchain-based smart contracts can be seen as a reification of such a concept, we think that their actual implementation may have hindered the full potential of the original idea. More precisely, as of now, smart contracts are implemented as *passive* entities, meaning that they actually enforce the terms of a physical contract only when "invoked" by one of their parties. As a consequence, smart contracts do *not* participate "actively" in the execution of a contract, since they do nothing until one of the parties sends a transaction to them. Furthermore, since they are deployed on the blockchain, their source code is *immutable*. Once deployed, they cannot be adjusted or adapted to changes that may occur in the real world.

Stemming from such considerations, we rethink smart contracts as *proactive* entities, that is, smart contracts able to *(i)* participate actively in the execution of a contract without necessarily waiting to be triggered through a transaction sent by its parties, *(ii)* interact *asynchronously* with each others, and *(iii)* automatically execute *postponed* or *periodic* computations. We then rely on logic programming to endow these smart contracts with a logic nature, in turns providing the capability of reasoning and planning in order to figure out the best course of action leading to a satisfactory achievement of theirs goals—i.e. the contractual terms. Moreover, such logic nature may allow some controllable mutability of the smart contracts behaviour – other than the capability of learning and exchanging behaviours by interacting with each others –, mitigating a well known issue affecting the mainstream notion of smart contract. In this thesis we model and implement a system for the execution of such smart contracts as a *proof-of-concept* for our idea, paving the way to the realization of *autonomous* smart contracts.

The reminder of this thesis is structured as follows. In chapter 2, we provide a technical background for what concerns blockchains, smart contracts, and logic programming. In chapter 3, we better describe our vision of *proactive* smart contracts, defining a set of requirements for the system we wish to realize. In chapter 4, we model the system and its components, describing their structure, interaction, and behaviour. In chapter 5, we provide an overview

on the system implementation, describing the main elements of interest. In chapter 6, we validate the system, providing examples of different use cases in which our smart contracts can be used to actively execute the terms of a contract. Finally, we discuss our smart contracts' autonomy in chapter 7, and conclude our work in chapter 8, presenting some interesting future works and directions.

Chapter 2

State of the art

In this chapter, we provide a technical background briefing on the arguments of interest of this thesis. In section 2.1, we describe blockchains, their functioning, and the methods that make their adoption viable in the context of distributed systems. In section 2.2, we examine the concept of *smart contract* along with two examples of real-word, *smart-contract-enabled* systems, namely Ethereum and Hyperledger Fabric. In section 2.3 we describe Tendermint, a novel general purpose consensus engine for blockchain applications, and we finally provide an introduction to logic programming in section 2.4.

2.1 The blockchain technology

Since their first appearance in 2008 with Bitcoin [13], blockchains have been gaining a lot of attention, especially in the research field. Nowadays, their practical usages range from entertainment to insurances, from supply chain management to healthcare. In this section we describe what blockchains are and how they work, highlighting strengths and weaknesses of this disruptive technology.

2.1.1 Data structure

At its core, a blockchain is essentially a data structure which stores information in blocks that are linked together, forming a chain. More specifically, each block stores an *hash pointer* to the previous block in the chain, except for the first one. An hash pointer consists of a pointer to where some information is stored together with the cryptographic hash (or *digest*) of that same information. So, to draw an analogy, a blockchain is like a linked list of blocks with hash pointers instead of normal pointers. Thanks to them, one can determine, for each block of the chain, the location of the previous block

and check through its digest if its content has not changed. The hash pointer which points to the most recent block is called *head*. To append new data to the blockchain, it is necessary to *(i)* create a new block which stores the data of interest along with an hash pointer to the most recent block and *(ii)* update the *head*, making it pointing to the newly block.

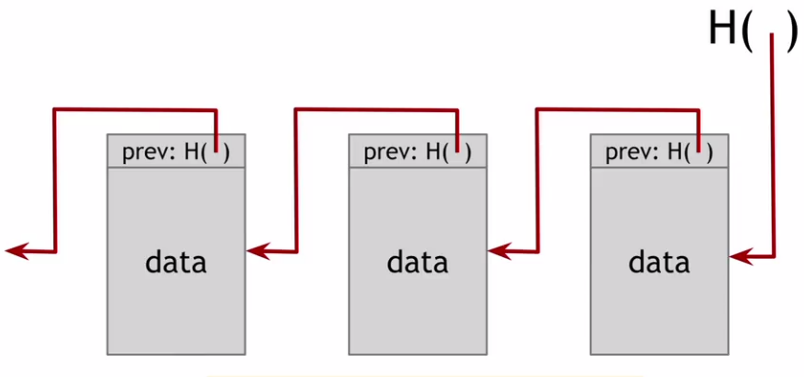


Figure 2.1: A blockchain. Each block is linked to the previous one in the chain by storing an hash pointer that points to it. Source: [26]

Hash pointers make the blockchain a valid candidate for *tamper-evident log* applications [26]. Let us suppose to have a blockchain shared by several users, each of them storing its personal copy of the blockchain *head*. If an attacker wants to tamper with data stored in the blockchain and tries to modify the content of a certain block *x*, then the digest of this block won't match up with the one stored in the hash pointer of block *x+1* anymore. To solve this discrepancy, the attacker will have update the latter, creating however a mismatch between the digest of block *x+1* and the one stored in the hash pointer of block *x+2*. Therefore, in order to preserve the integrity among hash pointers, even the smallest change made by the attacker in one block will unavoidably propagate to the *head* of the blockchain, making it possible for other blockchain users to detect the attack.

So far, the digest of a block has been defined as the cryptographic hash of the entire block. In practice, almost all existing blockchain technologies divide each block in two parts: the header, which contains meta information about the block (i.e. the *timestamp* of its creation *and* an hash pointer to the previous block), and the body, which contains the actual data to be stored. Such technologies define the digest of a block as the cryptographic hash of its header for efficiency reasons.

In order to keep the body data tamper-proof, another data structure which shares some affinities with blockchains is usually employed. This data structure is called *Merkle tree*, named after its inventor Ralph Merkle.

In its classical version, a Merkle tree is a binary tree. Let us suppose to split the content of a block body in several chunks. A merkle tree can be built out of these chunks as follows: the chunks, which make up the tree leaves, are grouped in pairs of two and, for each pair, a data structure containing two hash pointers, one to each of the chunks, is built. These data structures form the next level up the tree. At this point, they are in turn grouped in pairs to form the data structures of the upper level of the tree. This routine is repeated recursively until only one node is left. This node is the root of the tree, called *Merkle root*. Figure 2.2 shows a graphical representation of a Merkle tree.

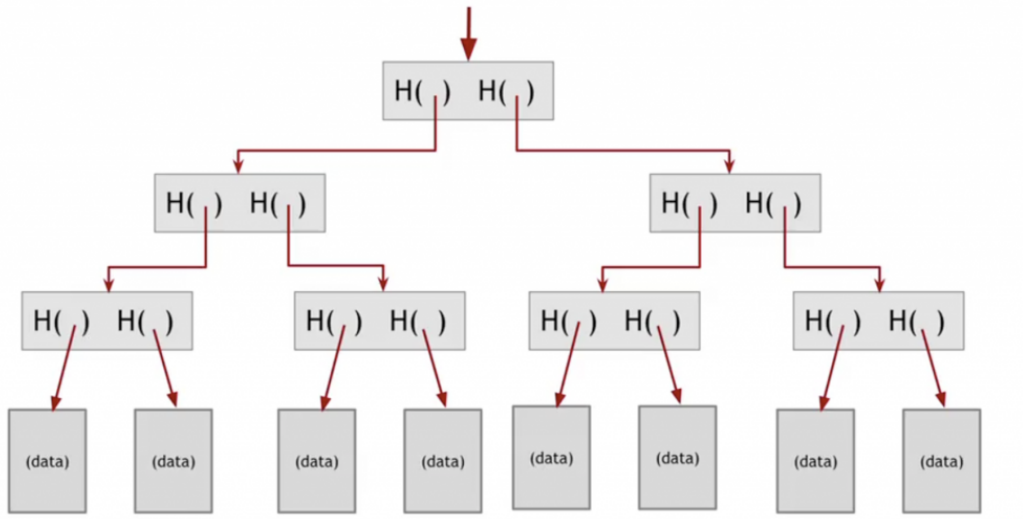


Figure 2.2: A binary Merkle tree. The leaves are grouped in pairs of two and the hash of each of them is stored in their parent node. The parent nodes are in turn grouped in pairs and their hashes are stored in the upper level of the tree. This routine continues until there is only one node left, that is, the root. Source: [26]

Building a Merkle tree from data chunks and storing the root in the block header prevents data chunks from being tampered with. This is because, as with the blockchain, if an attacker tampers with some data chunk at the bottom of the tree, the digest of the hash pointer that's one level up will not match anymore. If the attacker proceeds to tamper recursively with non-leaf nodes, the initial change will eventually propagate to the top of the tree, where he/she will not be able to tamper with the Merkle root that we stored in the block header.

Summing up, if someone tries to tamper with the Merkle root in the block header, the attempt will be detected thanks to the blockchain hash pointers. If someone tries to tamper with the data chunks of some block, the attempt will be detected thanks to the Merkle root stored in the block header.

Merkle roots has also a handful feature which blockchains lack: they allow for *proofs of membership*. If a blockchain user wants to prove that a data chunk is a leaf of a certain Merkle tree, all he/she has to provide is the set of nodes which are on the path between the data chunk and the Merkle root. The other nodes of the tree can be ignored. Other users can check the proof by calculating the digest of the given data chunk and verifying the hash chain of the nodes in the path up to the Merkle root.

Blockchains for themselves are great for those applications in which data is stored all in one place, being owned, managed and updated by a certificated entity. As long as the application users trust this entity, they can befit from the blockchain security with respect to the data stored in it. When data doesn't have an owner and it needs to be both replicated and shared by several mutually-untrusted users, giving each one of them the possibility to update it, a blockchain alone in not enough anymore. It is necessary to deal with a range of issues, which includes *(i)* identities management, *(ii)* how users can trust each other, *(iii)* who gets to propose a new block of data to be appended to the blockchain, and *(iv)* how users can reach consensus on the proposed blocks. We cover these issues in the next section.

2.1.2 Identities, trust and consensus

As the previous section provides a technical description of the blockchain as a data structure, we now explain how blockchains can be used in a distributed context.

Terminology and Functioning Overview

The expression "BlockChain Technologies" (or BCTs henceforth) refers to existing distributed applications which rely on a blockchain to provide their functionalities, such as Bitcoin. Systems of this kind usually consist of a network of nodes connected in a peer-to-peer fashion, each one of them storing a copy of the shared blockchain. Such copies are updated consistently over time thanks to an algorithm, which is employed by nodes to agree upon the next block that will be appended to each copy of the blockchain. This algorithm is called *consensus*; we provide a description about consensus algorithms in one of the following sections.

Users of these systems need to send data to one of the nodes in order to perform a writing operation on the blockchain: this data will be spread by the recipient node to its peers and eventually included in a new block, which will be appended consistently to each blockchain copy. Thus, the consensus algorithm ensures that all the *events* that occur in the system (i.e. the writing

operations) are seen and perceived in the same order by all nodes.

Furthermore, from the application standpoint, each node of the system has a *state*, which can be seen as the result of all the events that have occurred since system boot. Each new event leads to a state transition. Since the consensus algorithms ensures total ordering of all events, nodes share the same state and update it consistently over time.

As a result, every BCT is inherently a good example of *State Machine Replication* [14] where, given a state machine replicated over multiple nodes with each replica running in parallel with respect to one another, changes to a replica are propagated consistently to all other replicas.

Finally, for the sake of clarity, in this section we use the word "entity" to denote either a node or a user of the system.

Identities and Trust

Every BCT has its own methods for dealing with identities management, trust among entities and consensus on blocks. Some of these methods shares the same concepts, being slightly different in practice; others are totally different from one another. Their choice is strongly influenced by the type of blockchain that is adopted. Even if there is no clear definition, blockchains are usually categorized in one of the following types:

1. **Public or permission*less* blockchains**: this term is used to denote those blockchains where there are no restrictions on who can interact with them (i.e. reading/writing data). In other words, there isn't a trusted, third-party entity which controls the blockchain and regulates accesses to it. Examples of blockchain technologies which rely on a public blockchain are Bitcoin [13] and Ethereum [17].

2. **Private or permission*ed* blockchains**: as opposed to public ones, these blockchains are controlled by a third-party, federated entity which regulates accesses to them. Note that this entity could use an access policy which is more fine-grained than the "all or nothing" one. For example, a *role-based access control* (RBAC) policy [6] could be employed, granting full access to some entities and *read-only* access to others by means of roles. Examples of blockchain technologies which rely on a private blockchain are Hyperledger Fabric [34] and Corda [24].

As many other sorts of distributed system, BCTs assign an identity to each one of their entities for accountability purposes: only who is endowed with an identity can interact with them. The ones which rely on a public blockchain establishes how everyone can generate, on their own, an identity, since there

is not a third-party entity in charge of doing so. Conversely, with private blockchains identities are created and assigned only by the third-party entity that controls them, which acts as a Certification Authority (CA). Note that in this case the CA has total control over the number of entities which are allowed to interact with the system (i.e. users) or be a part of it (i.e. nodes), being able to set an hypothetical upper-bound to it.

In most cases, blockchains technologies use asymmetric cryptography to deal with both identity management and trust issues. Each entity is equipped with a self-generated pair of cryptographic keys and its identity is represented *(i)* by the public key, in the context of public blockchains[1], or *(ii)* by a certificate that is created and released by the CA, in the context of private blockchains. A certificate usually consists of the entity's public key, an expiration date, some optional data about the entity, and a signature made by the CA with its private key. Also, if a RBAC policy is adopted, a certificate shall contain even a role, which established the privileges of the certificate's owner within the system, that is, what operations the owner is allowed to do.

Trust issues can now be resolved as follows: to write some data on the blockchain, a user must first sign this data with its private key; every node or other user that will, respectively, receive or consult this data along with its signature (and eventually the user certificate) will be able to verify both authenticity and integrity of that same data. Likewise, every time a node proposes a new block to its peers, it signs the block with its private key in order to be trusted.

Finally, in order for nodes to reach consensus on the next block to be appended to the shared blockchain, some BCTs rely on one of the "classical" consensus algorithms, while others adopt different approaches. We cover this matter more specifically in the following section.

Consensus

In the field of distributed systems, consensus is a very-well known and studied problem that many applications have to face. With regard to this problem, the literature offers a broad range of algorithms that can be adopted as viable solutions.

Some of these algorithms, such as *Paxos* [8], introduce the concept of *voting*, being the following the idea behind them: each network node has the ability to vote for data (blocks in our case) proposed by other nodes; every time a proposal is made, consensus is reached through election: every node gets to vote on that proposal; if the number of gathered votes exceeds a prefixed *quorum*, every node proceeds to add the proposed data to its local database

[1] Actually, many blockchain technologies use a digest of the public key to identify entities.

(the blockchain in our case). Other consensus algorithms, such as *Raft* [19], rely on a *leader-based* approach: upon system start, a leader is chosen among the system nodes Its duty is to deal with proposals by following certain rules, with the purpose of reaching consensus on those proposals.

In addition to ensuring data consistency among replicas, some of these algorithms, such as *PBFT* [11], *BFT-SMaRt* [16], and *Honey Badger* [25], have been modelled to be *Byzantine fault tolerant* (BFT), that is, given n nodes involved in the consensus, they can tolerate up to $n/3 - 1$ of them exhibiting arbitrary behaviour. Further details and comparisons about BFT algorithms can be found in [15].

All of these "classical" algorithms can be adopted in real distributed systems and they are ideal when nodes participating in consensus are known *a priori*. Yet they present the following two weaknesses.

1. **Performance degradation**: as the number of nodes in the system starts to grow (i.e. > 100 or 1000 depending on the algorithm), most of these algorithms suffer from a performance degradation, which makes them unusable in practice [23]. In this context, with the term "performance" with respect to one of such algorithms we refer to the amount of data per second on which consensus can be reached (i.e. the throughput).

2. **One entity/multiple identities**: systems which rely on a quorum-based algorithm are vulnerable to the *Sybil attack* [12], since a single malicious entity could set up multiple nodes with different identities and subvert the election process. This is because he/she would be able to reach the quorum without the need of any further (honest) vote.

Summing up, such algorithms are not a good solution for open systems (like BCTs relying on a public blockchain) because of *(i)* the number of nodes participating in consensus, which can be arbitrarily high, and *(ii)* the lack of control on the relationships entity/identity. Nonetheless they are still ideal for reaching consensus in the context of private blockchains, where the number of nodes and their identities can be strictly monitored by the CA.

As an alternative solution, most of the existing BCTs which rely on a public blockchain adopt a *competition-based* approach: they employ "novel" consensus algorithms (with respect to the classical ones) where nodes compete against each other to be the first to propose a new, valid block. On the one hand, these algorithms are designed to be resistant to Sybil attacks while working with a virtually infinite number of nodes. On the other hand, they are far less performing than the classical ones in terms of throughput.

For the sake of brevity, in this section we provide a description for the most popular competition-based algorithm only, namely *Proof-of-Work*. Examples

of other competition-based consensus algorithm are *Proof of Stake* (PoS) [32], *Proof of Elapsed Time* [31], and *IOTA Tangle* [28], being [30] an excellent survey on them.

Proof of Work (PoW) was first proposed in [10] as a countermeasure to email-spam and denial of service attacks. In general, in a systems employing PoW, nodes need to solve a computational-expensive problem in order to send data to other nodes. This problem usually consists of a cryptographic puzzle related to the data that a node wishes to send. When a node finds a solution to it, it sends the data to the desired recipients along with this easy-to-verify solution, which represents the proof of its effort (hence the name "proof of work"). When receiving some data, a node acts as follows: if the related solution is valid, it accepts the data; if the related solution is invalid or missing, it ignores the data.

In the context of public blockchains, consensus on blocks can be reached by binding additions to the blockchain to proof of work contributions: in order for a node to append a new block to the chain, a cryptographic puzzle related to that block must be solved first. This approach makes the system resistant to Sybil attacks, since the number of valid block proposals (votes) that an attacker can make are no longer related to the number of its identities but rather to his/her computational power (called "hashpower" in this context). Furthermore, systems employing PoW as a consensus algorithm do not suffer from performance degradation implied by their number of nodes.

Let us explain this approach more in detail by referring to Bitcoin's PoW: in order to add a new block to the chain, a node has to repeatedly calculate the hash of this block, while varying a field of its known as "nonce" (i.e. a progressive number), until the output is lower than or equal to a certain *target* value. This repetitive process is called *mining*, while its actors are called *miners*. When a miner finally finds a nonce that gives a valid output, which makes up its proof of work, it proposes the block to its peers multicasting it to them. When receiving a block, a miner can easily verify if its hash results in a valid one (i.e. if it's lower than or equal to the target). If so, it includes the block in its copy of the blockchain and begins to mine on top of it.

The target value is set by the PoW algorithm and determines the average number of hashing operations that a node must do in order to obtain the desired output (i.e. the difficulty of the puzzle). Furthermore, the difficulty value is periodically adjusted by the algorithm itself in order to compensate the variation of computational power of the nodes composing the system. Note that, as long as a *strong* hash function is used, a node is not able to cheat, since the only way for it to obtain a solution to the problem is by means of *brute-force*, which is how required by the algorithm.

Following this approach, a blockchain fork can occur if two valid block

proposals are made by different nodes at about the same time. In order to solve this problem, miners follow a simple rule: in case of forks, they always choose (and mine on) the longest chain. Eventually, either the original chain or its fork will become longer than the other, making all nodes dropping the shorter one. Thus, the algorithm ensures *eventual-consistency* among nodes.

PoW is a great solution to the consensus problem for BCTs which rely on a public blockchain. Still, it has its own defects:

- **Electricity consumption**: as stated before, mining is a computationally-expensive process, meaning that it requires a lot of electricity. In a real system with thousand of miners, such as Bitcoin, the overall mining process results in a huge consumption of electricity.
- **Incentives**: there is no reason for miners to spend their computational resources if no reward is given when a solution to the puzzle is found. Thus, some kind of incentive is required in order for PoW to be usable in practice and for miners to keep behaving honestly.
- **The 51% attack**: the PoW consensus can be subverted if an attacker manages to gain control on a number of miners such that the sum of their computational powers exceed half of the total. This is because, in such scenario, the attacker would be able to mine new blocks at a faster pace with respect to honest miners. Thus, he/she could *(i)* create a blockchain fork on purpose, *(ii)* mine new blocks on top of it until its length exceeds the one of the "original" blockchain, and *(iii)* propose it to the honest miners, which would drop the original one. As a result, the attacker would have total control over the blockchain, being able to change its content as desired.

With regard to the problem of incentives, BCTs usually reward miners with some valuable asset whenever they find a solution to the cryptographic puzzle or whenever they correctly include data in a block on behalf of a certain user: in most cases, this asset is a digital currency. Furthermore, in most BCTs users send data to nodes through special messages, called *transactions*. Figure 2.3 depicts the employment of a blockchain in a distributed context.

Summing up, it is possible to rely on cryptography and consensus algorithms to make blockchains viable in the context of distributed systems. With each node of a system storing a copy of the blockchain and a consensus algorithm which ensure data consistency among such copies, one can benefit from both blockchains' properties (e.g. *immutability of the past*) and distributed systems' properties (e.g. availability and fault-tolerance). Nonetheless, many BCTs suffer from a common problem, that is, *scalability*. More precisely, the number of transactions per second which can be processed by a BCT is strictly

dependent on the transaction size, the block size, the latency among nodes, and the algorithm of consensus which is employed. It does *not* scale as system users increase. For a given BCT, if the number of transactions per second sent by users exceeds its throughput, some of these transaction will have to wait for some additional time (i.e. delay) before being included into a block. The more the number of transactions per second, the longer the delay.

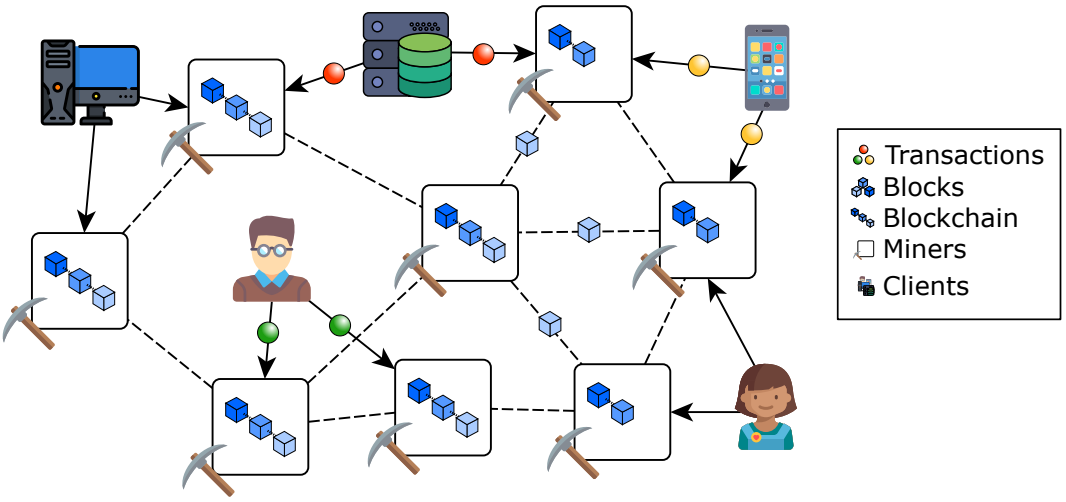


Figure 2.3: A blockchain in a distributed context. Each miner stores a copy of the blockchain and participates in the consensus protocol to consistently update it. Clients (i.e. end users) send transactions to miners to let them be included into a block.

2.1.3 Real-world usages and examples

We now briefly describe some applications that can be realized thanks to the usage of blockchain in a distributed context, along with some real-world example.

Cryptocurrencies : the most prominent use case up to now is the one of cryptocurrencies. Bitcoin and Ethereum are concrete examples of how a new digital and decentralized currency can be created relying on blockchains and PoW. The digital currency is decentralized in the sense that it is neither controlled nor released by a central authority, such as a bank, but rather generated by miners when a new block is correctly added to the blockchain. In this scenario, amounts of this currency are exchanged among end users by means of transactions; each block of the chain consists of a set of transactions. Therefore, the blockchain can be seen as a distributed ledger, which keeps track of all the money exchanges which have occurred within the system. The system state is

implicitly represented by the amount of currency that each system user currently holds, which is the result of all the transactions included in the blockchain. Thanks to the blockchain and PoW, users cannot revert a transactions made in the past nor *double-spend* the same transaction.

Distributed asset tracking : cryptocurrency-oriented BCTs represent a specific use case of distributed asset tracking applications, where the asset that needs to be tracked is the amount of currency hold by each user of the system. In general, there is no restriction on the nature of this asset, which could be, for example, an image or a textual file.

Digital Identities Management : many central institutions and companies stores the digital identities of their citizens/employees in a centralized way, that is, all in one place. Blockchains can be used as a novel approach to securely decentralize the management of digital identities. One example of company providing this kind of service is represented by OneName[2].

Online Voting Systems : the inherent transparency and security of public blockchains could be exploited to build an online voting system [27], in order to provide greater transparency in the voting process with each vote being securely stored on the blockchain. In 2015, the Bitcoin foundation started a project[3] with this exact goal.

2.2 Smart-contract-enabled BCTs

Through the past ten years, the blockchain technology has undergone considerable development which led to its increasing adoption in real distributed systems. This development process is usually split in phases. Even if there is not a clear distinction of these phases in the literature, they are usually three, which result in three so-called blockchain "generations" [22]:

1. **Bitcoin and Cryptocurrencies**: in the first generation, BCTs were seen mainly as a means to build a monetary system in which no central entity is involved in the management of the currency.

2. **Digital asset tracking and smart contracts**: as time went on, developers began to believe that a blockchain could do more than simply document transactions. Thus, the idea that digital assets of any kind and

[2] `https://onename.com/`

[3] `https://bitcoinfoundation.org/voting-on-the-blockchain-version-1-0/`

trust agreements could also benefit from blockchain management began to grow, leading to the adoption of smart contracts.

3. **The future**: we are currently in the third generation of blockchains, where developers study new BCTs to deal with existing BCTs' scalability issues. Examples of third-generation BCT are BitShares[4] and Steem[5].

In this section, we provide a definition of such *trust agreements* among users, known as smart contracts, and we describe them referring to the most well-known smart-contract-enabled BCT, namely Ethereum. Finally, we briefly explain the functioning of Hyperledger Fabric, another smart-contract-enabled BCT, in order to offer a comparison with respect to Ethereum.

2.2.1 Smart contracts

The concept of smart contract was first introduced by Nick Szabo in [5] as a "computerized transaction protocol that executes the terms of a contract". In other words, a smart contract is the digitalized version of a classic contract which has the tasks to *(i)* regulate the interaction between its parties while enforcing the terms of the contract, *(ii)* clarify the implications of this interaction, *(iii)* minimize exceptions, both malicious and accidental, and *(iv)* minimize the presence of trusted intermediaries. In Szabo's vision, smart contracts could be used as a means for reducing the transaction costs related to the execution of some kind of contracts as well as "opportunities for creating new kinds of businesses and social institutions".

Nowadays, such a concept has found its implementation in the context of cryptocurrency-oriented BCTs, being Ethereum the most mature and studied smart-contract-enabled BCT. In general, smart contracts are designed and implemented as computational processes which act as a means for executing general purpose code on top of a blockchain. With their actual implementation, smart contracts have the following properties:

- **User-defined**: smart contracts are created and deployed by end users.
- **Stateful**: each smart contract has its own internal state and exposes a set of methods finalized at changing it. Furthermore, smart contracts can usually take custody of assets on the blockchain.
- **Public**: the source code of the smart contract is stored on the blockchain, meaning that each end user is able to inspect it.

[4] `https://bitshares.org/`
[5] `https://steem.com/`

- **Immutable**: being deployed on the blockchain, the source code of smart contracts is immutable.
- **Replicated**: in a given moment, there are multiple copies of a given smart contract, one in each node which stores the blockchain. Nonetheless, the copies of a smart contracts always act as one.
- **Deterministic**: the execution of a smart contract is *always* deterministic. We explain this matter more in detail in subsection 2.2.2.
- **Reactive**: smart contracts are executed only when someone issues a transaction to them.

Before explaining them more in detail with Ethereum, let us observe what follows: there is a clear discrepancy, as greatly explained in [29], between the original idea behind smart contracts and their actual implementation in BCTs, since the latter is not strictly related to the classical concept of a contract. This is probably caused by their multifaceted nature, being it strictly related to both Law and Computer Science. Thus, experts of both fields should work together to be able to fully exploit the potential of the original idea employed in the context of BCTs.

2.2.2 Ethereum

Ethereum[6] is a cryptocurrency-oriented BCT which acts as platform for the deployment and execution of smart contracts [17]. The system relies on a public blockchain and consists of a set of nodes linked in a peer-to-peer network. Some of these nodes act as clients for end users to interact with the system. The others, called miners, store the blockchain and participate in a PoW consensus algorithm to consistently update it. They also take charge of executing smart contracts deployed by end users. For the effort put in the block mining process, miners are rewarded with some amount of *ether* - the native cryptocurrency of the system - whenever they mine a new block or execute a transaction. The blockchain is used for tracking the possessions of *ether* and storing both smart contracts' source code and end users' transactions.

End users are endowed with a pair of cryptographic keys and interact with smart contracts by publishing transactions. From their standpoint, the system is perceived as a single, consistent state machine, which coherently responds to all their inputs; thus, end users see every existing smart contract as a unique computational process; technically, this is not true: since smart contracts are deployed on the blockchain, each of them is actually replicated all over the

[6] `https://www.ethereum.org/`

network, with one copy in each miner. So, every time a user invokes the execution of a certain smart contract by publishing a transaction, all the copies of the target smart contract are invoked and all the same operations are performed, on all miners. This technical matter has two important implications: thanks to it, the overall system state is kept consistent among miners; because of it, the execution of smart contracts *must* be deterministic, otherwise miners would not be able to reach consensus on its result.

After this short overview, we provide a more detailed description of the system functioning in the following subsections.

Ethereum Virtual Machine (EVM)

Each miner is endowed with an interpreter for smart contracts, know as *Ethereum Virtual Machine* (EVM). The EVM bytecode set is *Turing-complete* and users can choose between two *contract-oriented* programming languages to write smart contracts with, namely *Serpent* (which is actually deprecated) and *Solidity*[7]. Programs written in such languages can be compiled down to EVM bytecode and thus be executed by miners.

More in detail, Serpent and Solidity are designed to write smart contracts whose execution is strictly deterministic. Thus, they don't provide the programmer with random number generators or any other randomness source. Nevertheless, programmers can still rely on some hard-to-predict data to simulate randomness, such as the hash of a block or its timestamp.

Accounts

In Ethereum, the application state is made up of data structures called *accounts*, while state transitions are caused by information exchange between these accounts, which occur when a end user publishes a transaction. Each account is associated with a fixed-length address, which identifies either a end user or a smart contract. Thus, there are two types of account: *externally owned accounts* (EOA), which belong to end users and are controlled by their private key, and *contract accounts*, which belong to smart contracts and are controlled by their source code. In the former case, the address is a cryptographic hash of the end user's public key; in the latter, the address is generated from a combination of the smart contract creator's address and a progressive number. An account stores the amount of ether possessed by its owner, a nonce used to avoid for a same transaction to be processed twice, the smart contract source code (if any), and the smart contract's internal storage (if any, initially empty). As a result, both end users and smart contracts can own

[7] `https://solidity.readthedocs.io/en/v0.4.24/`

ether. Figure 2.4 offers a graphical representation of both an EOA account and a contract account.

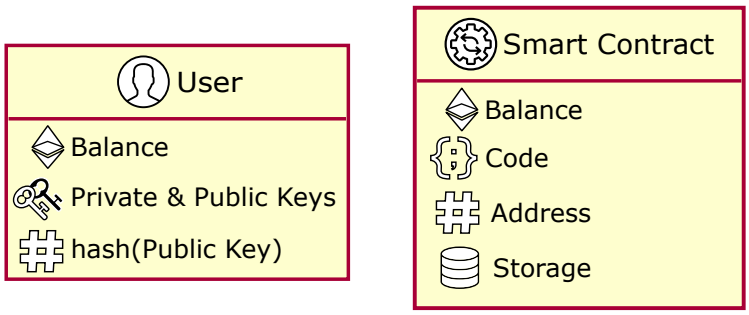


Figure 2.4: A high-level representation of two accounts. Nonces are omitted. Furthermore, The user's keys are depicted for clarity purpose only and are not actually included in the account data structure.

Transactions and Messages

Users are able to publish transactions in the system at any time. The act of publishing consists of multicasting a transaction to some miners, which proceed to validate it. If the transaction is valid, miners execute it and keep it in their mining pool along with other transactions. The transaction will be eventually included into a new block, on which consensus will be reached.

As depicted in Figure 2.5, there are three type of transactions: *(i) transfer* transactions, used to transfer a certain amount of ether from one account to another; *(ii) deployment* transactions, used to create a new smart contract; *(iii) invocation* transactions, used to invoke the execution of a smart contract. Regardless of the type, the transaction fields are always the same. These fields are:

- The **recipient** of the transaction (i.e. its address).
- A **cryptographic signature** of the transaction, made by the sender with his/her private key.
- A **nonce**, that is, a progressive number representing the number of transactions previously sent by the sender.
- The **amount of ether** to transfer from the sender's account to the recipient's one (possibly 0).
- An optional **data field**.
- Two additional fields called ***gasLimit*** and ***gasPrice***, described subsequently.

As a result, the type of a transaction can be deducted only by the value of its fields: deployment transactions have no recipient and their data field contains the source code of the smart contract to be created; transfer transactions have an EOA address in their recipient field and the data field left empty; invocation transactions have a smart contract address in their recipient field with some optional input parameters in their data field.

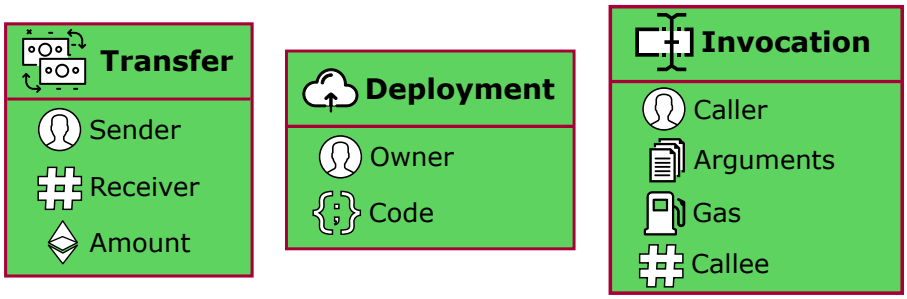


Figure 2.5: A high-level representation of the three types of transaction in Ethereum.

Smart contracts are not able to publish transactions, since they are not endowed with a private key. Nevertheless, they can still send *messages* to other smart contracts. A message is much like a transactions: they share the same fields except for the signature field and the gasPrice field, which are not included in the former. Every time a smart contract sends a message to another one, it waits until a result is received or an exception is raised. Thus, the interaction between smart contracts is always synchronous.

Miners and consensus

In Ethereum, miners are responsible for *(i)* validating transactions published by end users and spreading them to their peers subsequently by means of *gossip*, *(ii)* participating in the consensus process in order to mine new blocks, and *(iii)* executing transactions.

- **Validation**: a transaction is considered to be valid when it's well formed, its signature is authentic, its nonce coincides with the total number of transactions that the issuer has previously published and the amount of ether which the issuer wishes to transfer (if any) doesn't exceed his/her balance.
- **Consensus**: the consensus algorithm engaged by miners is a variant of the GHOST protocol [33]; thanks to it, a new block is generated once every 15 seconds on average[8].

[8] `https://etherscan.io/chart/blocktime`

- **Execution**: transactions are executed by a node after the validation process or when a block is received; the execution of each transaction causes a state transition.

The Ethereum pricing model

The *gasLimit* and *gasPrice* fields are required in invocation transactions in order adhere to the system anti-DoS (Denial of Service) model: Ethereum introduces the concept of *gas* as the fundamental unit of computation. In other words, each computational step performed by miners costs 1 gas. The gasLimit field represents how many computational steps the transaction execution is allowed to take; the gasPrice field represents the fee which the sender is willing to pay for each computational step. During the actual transaction execution, if the number of computational steps exceeds the gasLimit value, an exception is raised and all the side effects of the execution are reverted. In any case, upon termination, the whole computation cost is computed (gasPrice · computational steps taken) and withdrawn from the balance of the transaction issuer. This amount of ether is redeemed by the miner which includes the transaction in a new block as a reward for its computational effort. Actually, some computational steps (i.e. bytecode instructions) cost more than 1 gas, since they are more computationally expensive. A full pricing list can be found in [21].

This approach is adopted to prevent both accidental and hostile infinite loops, which would otherwise break the system. In fact, if transactions were able to trigger infinite computation, a end user may easily succeed in a Denial of Service attack to the whole Ethereum network by simply deploying and then invoking some non-terminating smart contract.

Note that miners usually privilege the execution of transactions with higher gasPrice in order to obtain a greater reward. Thus, end users can adjust their transactions' gasPrice depending on their needs: they can set a higher gasPrice if they want a high chance for their transaction to be executed almost immediately, or set a lower gasPrice otherwise.

2.2.3 Hyperledger Fabric

Hyperledger Fabric[9] is a modular and extensible open-source system for deploying and operating permissioned blockchains, which supports the execution of distributed applications written in any programming language [34]. It introduces a novel blockchain architecture aimed at enhancing resiliency, flexibility, scalability, and confidentiality.

[9] `https://www.hyperledger.org/projects/fabric`

Most of the existing BCTs rely on the so-called *order-execute* architecture: a consensus protocol is adopted by peers to order transactions and then each peer executes all the ordered transactions sequentially. This architecture suffer from several limitations: *(i)* the consensus protocol is *hard-coded* within the system and cannot be changed according to necessities; *(ii)* the validity of transactions is established by the consensus protocol and cannot be adapted to the application requirements; *(iii)* within smart-contract-enabled BCTs, smart contracts must be written in a *domain-specific* language, which limits the adoption of the system as it requires extra effort to learn that language; *(iv)* the sequential execution of transactions by all peers limits performance.

To overcome these problems, Fabric introduces the *execute-order-validate* architecture by separating the transaction flow in three phases, which are performed by different entities within the system.

Fabric Network

In order to adhere to this architecture, a Fabric network is usually composed by the following modules:

- **Membership service provider**: this module plays the role of the third-party entity (i.e. the CA) which deals with identity management within the system. Thus, it is responsible for associating peers with identities, maintaining the permissioned nature of Fabric.
- **Ordering service**: this module is in charge of establishing consensus on the order of transactions and broadcasting state updates to peers. Thus, it ensures a total order on all transactions. It consists of a set of nodes and each one of them is called an *ordering service node*.
- **Peer-to-peer gossip service**: its task is to disseminate the updates published by the ordering service to all peers.

Figure 2.6 depicts a standard Fabric network. Each peer locally stores the blockchain and the application state is represented through a key-value store. Furthermore, in Fabric it is possible to create and deploy smart contracts to handle application logic: they are called *chaincodes* and run in isolated environments (i.e. Docker containers). Each chaincode has to specify an *endorsement policy*, which must be fulfilled by transactions addressed to it in order for them to be actually executed. There is a concise set of endorsement policies from which a chaincode can choose from. In the general case, chaincodes must specify which or how many peers should endorse the transaction in order for it to be considered valid.

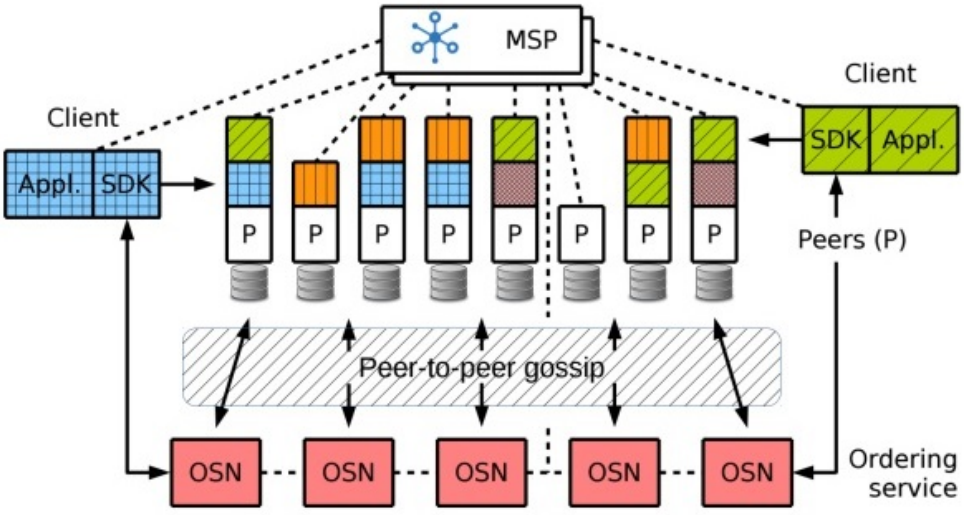


Figure 2.6: A standard Fabric network. Nodes in the network play one of three roles: client, peer or ordering service node. Their identities are managed by a federated membership service provider. Each peer runs multiple chaincodes, represented by the differently shaded and colored boxes. Source: [34]

Transaction Flow

The adoption of the *execute-order-validate* architecture splits the transaction flow in the three following phases:

- **Execution phase**: when an client wishes to interact with a chaincode, it sign and send a *transaction proposal* to all the peers specified by the chaincode's endorsement policy. These peers are called *endorser nodes*. Upon receipt of a proposal, an endorser locally executes it, checking its correctness. This execution is actually a *simulation* and does not persistently update the application state. When the simulation ends, the endorser signs an *endorsement* message related to the received proposal, which contains data about the simulation and its outcome, and sends it back to the end-user enveloped in a *proposal response*.

  The client collects endorsements until the endorsement policy of the desired chaincode is satisfied. Then, he/she assembles a proper *transaction* and sends it to the ordering service. A transaction is made up with a payload, transaction metadata and a set of endorsement messages.

- **Ordering phase**: in these phase, the ordering service nodes engage consensus to establish a total order on the submitted transactions, batching them into blocks.

- **Validation phase**: blocks are delivered to peers either by the ordering service or thanks to the peer-to-peer gossip service. Upon receipt, a block enters the validation phase. First, all the endorsement policies of its transactions are evaluated in *parallel*: if the endorsement is not satisfied, the corresponding transaction is marked as invalid and its effects are not applied. Subsequently, the block is appended to the local blockchain and all the valid transactions are executed. Their execution results in a state update.

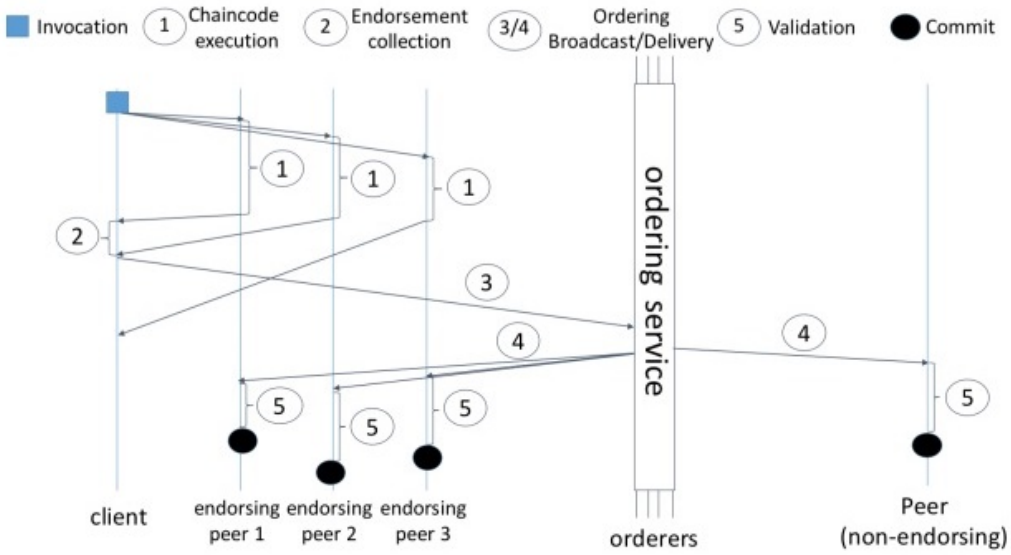


Figure 2.7: Fabric transaction flow. The client issues a transaction proposal related to a desired chaincode to one or more endorsing peers. They simulate the execution of the proposal and reply with endorsement messages. When the client has collected enough endorsements to satisfy the endorsement policy imposed by the chaincode, he/she creates a proper transaction and sends it to the ordering service. The transaction is included into a block, which is delivered to all peers. Finally, peers validate the block and add it to their local copy of the blockchain. Source: [34]

Advantages

The adoption of this modular architecture brings several advantages: *(i)* the consensus protocol is *pluggable*, since it is not hard-coded into the system; as a result, developers can choose the most suitable consensus protocol for their Fabric system and let the ordering service use it; *(ii)* the transactions execution occurs in parallel in the validation process, which makes Fabric reach a throughput of more than 3500 transactions per second; *(iii)* chaincodes run

in Docker containers and thus can be written in *any* programming language; *(iv)* the validity of each single transaction is determined by the chaincode endorsement policy to which it is addressed; this is in contrast with the one-for-all *trust model* adopted by other BCTs, such as Ethereum, which could be too restrictive for some kind of applications.

2.3 Tendermint

Tendermint[10] was introduced in 2014 by Jae Kwon as an alternative solution to the blockchain consensus problem [18]. At the time, the core idea behind it was to provide a consensus protocol which required no proof of work involved in the mining process and had a high level of protection against double spending attacks. In its very first version, the Tendermint software had a simple currency built in: to participate in consensus, its users had to bond units of this currency into a security deposit, which would be revoked if they misbehaved, making Tendermint a Proof-of-Stake algorithm [32].

Through time, Tendermint has evolved to become a general purpose, *quorum*-based consensus engine. Thus, it is not a fully-operating BCT but rather a "minimal" core, based on a private blockchain, which developers use to write their own application.

More specifically, the engine can host arbitrary applications: developers can write an application, in any programming language, and use Tendermint to replicate it over several machines in a secure and consistent way, creating their own BCT. In this scenario, every machine runs a replica of the Tendermint software and a replica of the application. Tendermint is "secure" in the sense that it can tolerate up to 1/3 of machines fail in arbitrary ways (i.e. it is Byzantine Fault Tolerant). It is "consistent" in the sense that every non-faulty machine sees all the transactions in the same order and computes the same application state. Note that Tendermint does not impose a fixed representation of the application state and developers are therefore free to represent it as they prefer (e.g set of accounts as in Ethereum, key-value store as in Fabric, ...).

The Tendermint software is made up of two components: a consensus engine, called *Tendermint Core*, and an application interface, called *Application BlockChain Interface* (ABCI). The consensus engine deals with all the aspects related to blockchain management and peer-to-peer connectivity, delegating the business logic to the application which developers build on top of it. It also exposes a set of remote procedure calls (RPCs) which allows end users to send transactions to it or to query the blockchain/application state. The application interface establishes a fixed set of messages which Tendermint Core

[10] `https://tendermint.com/`

and the application must use to interact with each other. This approach is in contrast with the one adopted in other blockchain technologies, such as Bitcoin and Ethereum, where all the aspects of a decentralized ledger are handled by a single program. Thanks to that, it is even possible to use Tendermint as a replacement for other blockchain consensus engines, as Tendermint's developers did for *Ethermint*[11] (Ethereum over Tendermint).

2.3.1 Tendermint Core

Besides the remote procedure calls, Tendermint Core is a Byzantine Fault Tolerant consensus algorithm. Before describing it, let us take a step back: in a system employing Tendermint, there is a set of *nodes* that are connected to each other in a peer-to-peer network. These nodes relay new information by *gossip* and each node keeps a complete copy of a totally ordered sequence of the events (i.e. transactions) which occurred in the system. This copy is stored in the form of a blockchain and the consensus protocol is used to update this shared blockchain in a consistent way among nodes. From the nodes point of view, transactions have no syntax nor semantics: they are arbitrarily-long byte arrays. It is a duty of the ABCI application to define the transactions structure and encoding, and therefore their syntax and semantics.

Each block has its own height (i.e. the number of blocks before it in the chain) and consists of a header and a body. The header contains metadata about the block and the consensus, as the number of transactions which composes its body, the block height, the block creation time, and the hash of the previous block. It also stores the transactions result returned by the application via ABCI and some additional hashes which can be used to verify the identity of nodes that participated in its commitment in the consensus. The body of a block simply stores all the transactions included in the block along with the set of votes that committed the previous block.

A node which participates in the consensus is called a *validator* and is endowed with a pair of keys. These keys give the validator an identity and allow it to send signed messages (called *votes* henceforth) to its peers, as the algorithm requires. The votes that a validator can create and broadcast are *pre-vote*, *pre-commit*, and *commit*. A validator has also its own *mempool*, where it can store all the transactions sent to it which wait to be included into a block.

The algorithm is based on a modified version of the DLS protocol [4] and is used among validators to reach consensus on new blocks. It is round-based: each round is composed by three steps (i.e. *Propose*, *Pre-vote*, and *Pre-commit*) along with two special steps (i.e. *Commit* and *NewHeight*). For

[11] `https://ethermint.zone/`

the commitment of the block at height H, there may be more than one round required to commit that block.

In the beginning of each round, a *proposer* for the new block is selected from the set of validators in a round-robin fashion. At this point, the current round goes through the following steps:

1. `Propose`: the proposer of this round broadcasts a proposal containing the new block. If the proposer is locked on a block from some prior round, it proposes that block along with a *proof-of-lock*. Proof-of-locks are created, and thus explained, in the pre-commit step.

2. `Pre-vote`: in the beginning of this step each validator makes a decision. If the validator was locked on a proposed block form some prior round, it signs and broadcast a pre-vote for that block. Otherwise if the validator had received an acceptable proposal for the current round, then it signs and broadcasts a pre-vote for the proposed block. If the validator had received no proposal or an invalid one, it signs a special *nil* pre-vote. No locking happens during the pre-vote step.

3. `Pre-commit`: at this point, if a validator had received more than the 2/3 of pre-votes for a particular block, it signs and broadcasts a pre-commit for that block. It also locks onto that block, releasing any prior locks. A node has a lock on at most one block at a time. If the node had received more than 2/3 of nil pre-votes then it simply unlocks. When locking (or unlocking) the node gathers the pre-votes for the locked block (or the prevotes for nil) and packages them into a *proof-of-lock* for later when it is its turn to propose. If a node had not received more than 2/3 of pre-votes for a particular block (or nil), then it does not sign or lock anything.

   At the end of the pre-commit step each validator makes a decision. If the validator had received more than 2/3 of pre-commits for a particular block, then the node enters the commit step. Otherwise it continues onto the propose step of the next round.

4. `Commit`: in this step the block is committed and the round is finalized, provided that two parallel conditions are both satisfied. First, the validator that enters this step must receive the block committed by the network if it had not already. Once the block is received, it signs and broadcasts a commit for that block. Second, the validator must wait until it receive at least 2/3 of commits for the block pre-committed by the network. Once both conditions are satisfied the validator sets its CommitTime to the current time and switches to the New Height step.

5. `NewHeight`: before starting the new round at height H, validators enter this step and waits a fixed amount of time past their local CommitTime. This wait is meant to gather additional commits for the just committed block at height $H - 1$, letting block proposals to include more than the minimum 2/3 of commits, thus allowing the commits of slower validators to be included in the blockchain.

At any time during the consensus process, if a node receives more than 2/3 of commits for a particular block, it immediately enters the Commit step if it had not already. Also, at any time during the consensus process, if a node is locked on a block from round R but receives a proof-of-lock for a round R' where $R < R'$, the node unlocks. The diagram in Figure 2.8 summarizes the algorithm described above.

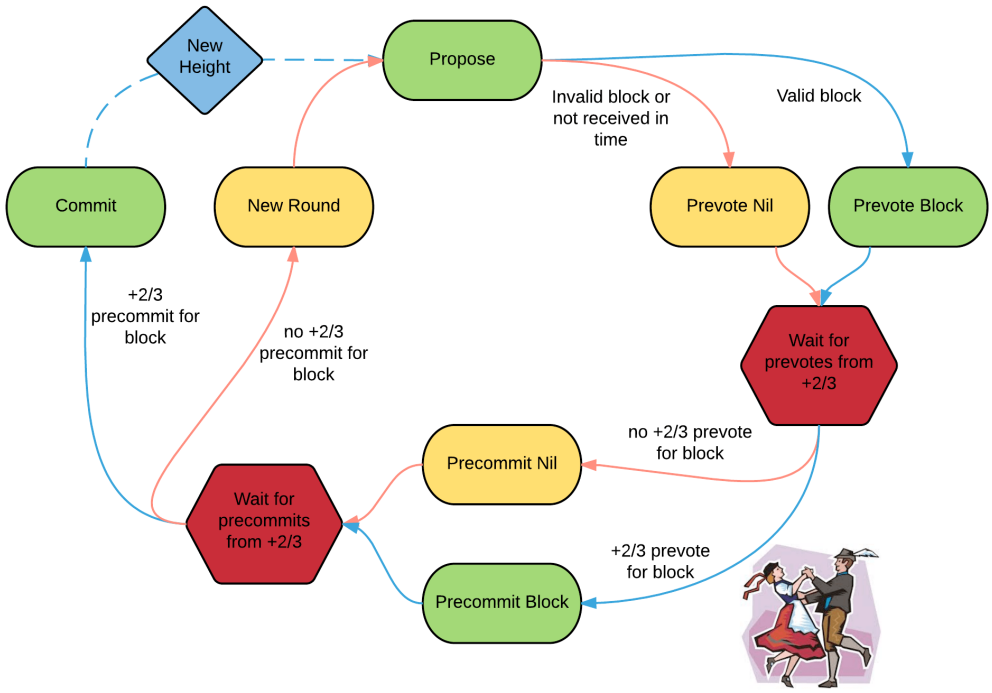


Figure 2.8: The Tendermint Core consensus protocol without locking rules. There is a picture of a couple doing the polka: in the Tendermint jargon a *polka* is when more than two-thirds of the validators pre-vote for the same block. Source: [20]

The Tendermint Core is Byzantine Fault Tolerant, which is what it makes it "secure". By default, all validators have the same voting power, meaning that all the votes created and broadcast have the same "weight" in the consensus protocol. This could be problem for those systems where there is the need of

different kinds of validators, which are required to have different voting power. Unfortunately, Tendermint isn't endowed with its own currency anymore, but since it can replicate arbitrary applications, it is possible for the developer to define his/her own currency, and denominate the voting power in that currency.

2.3.2 Application BlockChain Interface (ABCI)

The Application BlockChain Interface allows for Byzantine Fault Tolerant replication of applications written in any programming language. It essentially marks the boundary between Tendermint Core (called *core* henceforth) and the application written by the developer. The interface defines a fixed set of messages that the core, acting as a client, sends to the application when some event occurs (e.g. receipt of a new transaction, commitment of a new block, ...) and a set of replies that the application, acting as a server, is meant to send back to the core.

The primary messages that gets delivered from the core to the application are:

- **InitChain**: this is a message sent to the application upon system start. It contains information about the validators that will participate in the consensus protocol.

- **CheckTx**: message meant for validation purpose only. The core sends this message to the application upon the receipt of a new transaction, including the transaction in it. The duty of the application is to validate the transaction against the actual application state according to some internal logic. For example, the validation of a transaction could include the check for the progressive number (if any) of the transaction, and return an error if that number is not the one expected. If the core receives a positive reply to a CheckTx message, it proceeds to broadcast the corresponding transaction to its peers. Otherwise, the transaction is discarded. The application should *not* update the current state upon the receipt of a CheckTx message.

- **DeliverTx**: message meant to deliver a transaction to the application which has been successfully included into a block. For security reasons, the application should re-check the validity of the transaction before actually executing it. With the expression "executing a transaction", we mean updating the current state of the application.

- **Commit**: the commit message is sent from the core to the application when the former needs a cryptographic hash of the current application state, which will be included in the next block header. Thanks to this,

inconsistencies in updating the application states among the replicas will appear at core level as a blockchain forks, making the validators aware that something went wrong.

- **Query**: given a system employing Tendermint, an end user of such system can query the core of a certain node to retrieve information about the application state. When this happens, the core creates and sends a Query message to the application, which replies with the desired information.

Figure 2.9 summarizes the interaction between core and application dictated by ABCI. There are other secondary messages which the core is allowed to send to the application. A full list, along with a meticulous description, can be found on Tendermint's website[12].

Tendermint ABCI comes with its primary implementation, the *Tendermint Socket Protocol*. By default, it opens three different TCP connections form the core to the application, called *Mempool Connection*, *Consensus Connection*, and *Query Connection*.

- **Mempool Connection**: this connection is used only for CheckTx requests. Transactions, enveloped in CheckTx messages, are sent to the application in the same order they were received by the core.

- **Consensus Connection**: once a new block is committed, a series of messages is sent through this connection. First, a BeginBlock message is sent to the application, which contains information about the new block (e.g.: height, previous block hash, ...). Then a series of DeliverTx messages is sent, one for each transaction in the committed block, in order. Note that these messages are received by the application in the same order they were sent thanks to TCP. This series is followed by a EndBlock message and, finally, by a Commit message.

- **Query Connection**: this connection is used to query the application without engaging consensus. Queries sent from users to the core result in Query messages that are sent to the application through this connection.

[12] `https://tendermint.com/docs/spec/abci/abci.html`

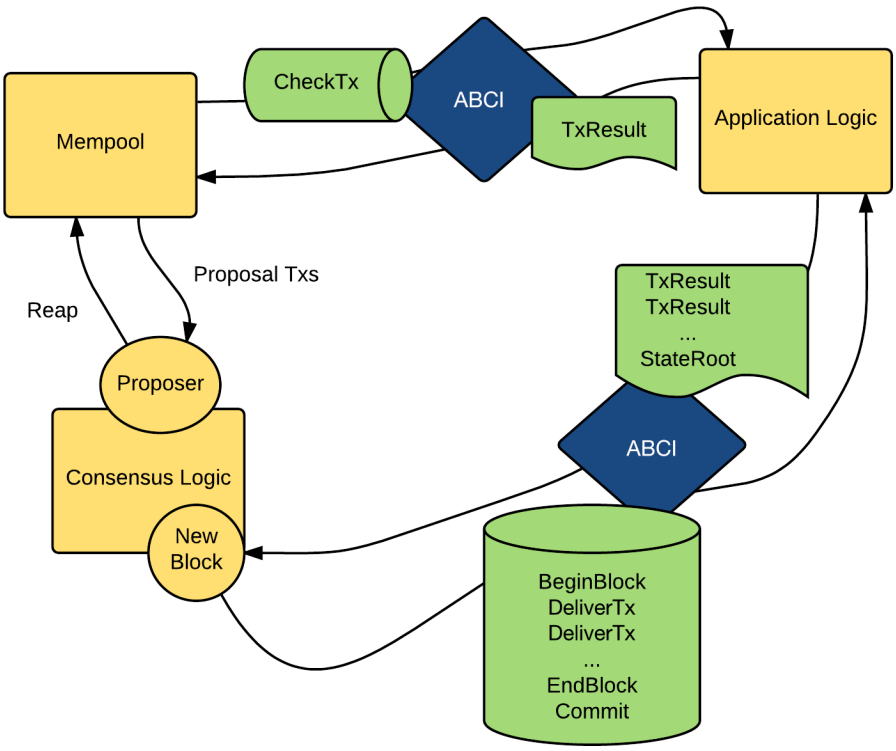


Figure 2.9: The core, represented by blocks "Mempool" and "Consensus Logic", interacts with the application, represented by the block "Application Logic", through ABCI. Source: [20]

Note that, at application level, transaction execution must be deterministic. Otherwise, different replicas of the application would reply to Commit messages with different hashes of the actual application state, breaking the system consistency. As a consequence, consensus would not be reached among the core replicas.

Summing up, thanks to this interface developers can write, in a language of their choice, their own server which follows the ABCI specifications and then use it to write an application which runs on top of Tendermint. Besides the one written in Go by Tendermint's developers, there are already other ABCI servers available, written in Python[13], Javascript[14], C++[15] and Java[16].

[13] `https://github.com/tendermint/tendermint/tree/develop/abci/example/python3/abci`

[14] `https://github.com/tendermint/js-abci`

[15] `https://github.com/block-finance/cpp-abci`

[16] `https://github.com/jTendermint/jabci`

2.4 Logic programming

Logic programming is the outcome of decades of research concerning how *inference*, the creative process we use to reason and draw conclusions, can be reproduced and automated with *computation*, a mechanical process where no creativity is required.

It is a programming paradigm based on formal logic which is rather different from others for the concept of computation in the first place: for a given input (i.e. an expression), computation usually consists in calculating the corresponding result following a set of fixed rules; within logic programming, for a given input (i.e. a conjecture), computation consists in searching a proof to that input in a solution space, following a fixed strategy.

In this section, we provide an introduction to logic programming and show how smart contracts could benefit from it.

2.4.1 The paradigm

Logic programming has its root in automated deduction and first-order logic. The former was studied by Kurt Gödel and Jacques Herbrand in the 1930s, whose work can be seen as the origin of the "computation as deduction" paradigm. The latter was first introduced by Gottlob Frege and subsequently modified by Giuseppe Peano and Bertrand Russell throughout the second half of the 19th century.

These works led Alan Robinson, in 1965, to the invention of a resolution principle based on the notion of *unification* [1], which makes it possible to prove theorems of first-order logic and thus compute with logic. The final steps towards logic programming were made in the 1970s by *(i)* Robert Kowalski, who introduced the notion of logic programs with a restricted form of resolution, compared to the one proposed by Robinson, as a feasible proof search strategy [2]; *(ii)* Alain Colmerauer and its team, who worked on a practical realization of the idea of logic programs, giving birth to Prolog.

The Logic programming paradigm can be summarized by the following three features [9]:

1. **Terms**: Computing takes place over the domain of all terms defined over a "universal" alphabet.

2. **MGU**: Values are assigned to variables by means of automatically-generated substitutions, called *most general unifiers*. These values may contain variables, called *logical variables*.

3. **Backtracking**: The control is provided by a single mechanism, called *automatic backtracking*.

The universal alphabet is composed by *variables*, *function symbols* (or *functors*), *parenthesis*, and the *comma* symbol. A *term* is defined as follows: *(i)* a variable is a term; *(ii)* a functor with arity 0, called *constant*, is a term; *(iii)* if `f` is a n-ary functor (or functor of *arity* n) and t_1,...,t_n are terms, then `f(`t_1`,...,`t_n`)` is a term. Terms which include no variables are *ground terms*; variables and constants are *atomic terms*, while terms built out of functors are *structured terms*. In general, terms have no *a priori* meaning: each one of them can be associated to a domain-specific entity by means of *pre-interpretation*.

Let us explain terms with the following examples: the variable `X` is an atomic term; the constant `a` is a atomic and ground term; given the functor `f` of arity 2, `f(a,X)` is a structured term; given another functor `g` of arity 3, `g(X, f(a, X), a)` is also a structured term.

Variables are non-initialised at first and they can be assigned one of all the possible terms. Assignments of a term to a variable are called *substitutions*. The notation:

$$\{X_1/t_1, \ldots, X_n/t_n\}$$

denotes a substitution, which assigns term t_i to variable X_i, for $1 \leq \text{i} \leq \text{n}$. Referring to the examples above, if we apply the substitution $\{X/t(c, Y)\}$ to term `g(X, f(a, X), a)`, we obtain `g(t(c, Y), f(a, t(c,Y)), a)`.

Given two terms, a substitution which makes them equal (if any) is called *unifier*. Thus, in logic programming, *unification* is the process of solving an equation between terms. This process adheres to a simple set of rules: *(i)* two non-initialised variables X and Y unify with the substitution X/Y; *(ii)* two constants unify if and only if they are the same constant; *(iii)* two structured terms unify if and only if they have the same functor and arity, and their terms unify recursively.

The least constraining unifier is called *most general unifier* (mgu). For example, given the expression

$$g(a, Y) = g(X, Z)$$

the substitution $\{X/a, Y/Z\}$ represents the *mgu* and is less constraining than substitutions $\{X/a, Y/b, Z/b\}$ and $\{X/a, Y/a, Z/a\}$.

In logic programming, atomic actions are equations between terms, which are executed through the unification process. In other words, the unification process is the basic operation on which the proof-search strategy relies to check whether a proof to a given conjecture has been found or not. As stated before, the proof-search strategy employed in logic programming is the one proposed by Kowalski, known as *SLD-resolution* principle. So, in order to understand how computation works in its entirely, in the following section we briefly explain how the SLD-resolution principle works, after a short but necessary introduction to *Horn Clauses*.

2.4.2 Horn clauses and the SLD-resolution principle

In logic, sentences are called *prepositions*, which can be written through *predicates*. Predicates are essentially structured terms and, in logic, are called *atoms*: if `p` is a predicate symbol of arity n and t_1,...,t_n are terms, then `p(`t_1`,...,`t_n`)` is an atom.

If the literal A is an atom then the logical formula "A" states that A is true, while "¬A" states that A is false. Literals can be combined through *logical connectives* to build more complex logic formulas. If A and B are literals, they can be combines through : *conjunction* (e.g. A ∧ B, both A and B are true); *disjunction* (e.g. A ∨ B, either A or B is true); *implication* (e.g. A → B, if A is true then B is true); *equivalence* (e.g. A ↔ B, A is true if and only if B is true).

A *logic clause* is a finite disjunction of literals. Given n literals A_1,...,A_n and m literals B_1,...,B_m, the formula

$$A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$$

is a logic clause with n "positive" literals and m "negative" literals, which is often written as

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

A conjunction of logic clauses is called a *Clausal Normal Form* (CNF).

Finally, *Horn clauses* are logic clauses with at most one positive literal. One example of them is represented by *definite clauses*, which are logic clauses with exactly one positive literal (e.g. $A \leftarrow B_1, \ldots, B_m$). Furthermore, there are two kinds of definite clauses: *unitary clauses*, which have no negative literals (e.g. $A \leftarrow$) and *definite goals*, which have no positive literals (e.g. $\leftarrow B_1, \ldots, B_m$). Both of them are Horn clauses too.

We are interested in Horn clauses since the SLD-resolution principle works as follows: given a logic program P written as a CNF of Horn clauses and a formula F, it shows that it is possible to compute (by contradiction) whether P *logically entails* F. So, in order to exploit it, logic programs are written as CNF of Horn clauses. In a logic program, definite clauses are called *rules*, unitary clauses are called *facts* and definite goals are simply called *goals*. Rules and facts make up the "source code" of a logic program, while goals represents the input that must be proven.

The SLD-resolution principle, as the one proposed by Robinson, proceed by *contradiction*: it negates the formula F and succeeds if it fails to prove it against the program P. To prove a goal G with respect to a program P, the principle works as follows:

1. It looks for a logic clause in P whose head (i.e. the positive literal) unifies with G.

2. There are three possible outcomes to this search:

   2.1 $\rightarrow$ no clause could be found: in this case the resolution fails.

   2.2 $\rightarrow$ a rule R of form $A \leftarrow B_1, \ldots, B_m$ is found: being θ the mgu of G and R, then the proof of G succeeds, and is represented by Gθ, if it is possible to further prove the sub-goals $B_1\theta \ldots B_n\theta$, where $B_i\theta$ is the application of θ to B_i. Thus, these sub-goals represent now the current goal.

   2.3 $\rightarrow$ a fact F is found: being θ the mgu of G and F, no sub-goals are added to the current goal and the solution is represented by Fθ.

3. If the current goal is empty, the resolution ends successfully (SLD refutation). Otherwise, a *selection rule* is adopted to choose the next sub-goal to prove, starting back from point 1. If the current goal never gets emptied, the resolution does not terminate.

Furthermore, resolution relies on *automatic backtracking*: at a given point τ in the resolution process, for the current goal/sub-goal G there could be more than one clause in the program whose head matches with it; after choosing one of them, if the resolution of the related sub-goals fails, the process automatically backtracks to point τ, where another clause is chosen. Backtracking is performed until at least one "candidate" clause is still present.

The SLD-resolution principle is often called a "proof search" because of its non-deterministic nature: as just stated, while looking for a clause in the program P, there could be more that one whose head matches with the current goal (*or-nondeterminism*); furthermore, non-determinism occurs when the resolution process has to choose the next goal to prove among several sub-goals (*and-nondeterminism*). The way these forms of non-determinism are dealt with represents how the proof-search proceeds in the solution space.

2.4.3 Prolog, strengths and weaknesses

Prolog is a programming language which allows to write logic programs. An example of logic program is as follows:

```
% Some facts.
% manuela is a parent of alfredo
% manuela is a parent of laura
% ...
parent(manuela, alfredo).
parent(manuela, giovanni).
parent(dino, manuela).
parent(laura, manuela).
```

```
% A rule.
% X is a grandparent of Z if X is a parent of
% some Y AND this Y is a parent of Z.
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

Listing 2.1: An example of logic program in Prolog. Lines starting with "%" are comments. The implication symbol (←) is replaced by the symbols ":-" and the syntax requires for each clause to be ended with a *period* (i.e the character "."). Recall that clauses have no a priori meaning attached, but one can assign a meaning to them by pre-interpretation.

There are several implementations of these language, such as SWI-prolog[17] and tuProlog [18]. They serve as *logic engines*, allowing programmers to write logic programs and submit goals to them. When a goal is submitted, the SLD-resolution process starts, resulting in a goal proof or a resolution failure. For example, submitting the goal `parent(dino, X)` to the above program using tuProlog gives the following output:

```
yes.
X / manuela
Solution: parent(dino,manuela)
```

Listing 2.2: The solution to goal `parent(dino, X)` submitted to the logic program displayed in Listing 2.1. The goal can be interpreted as "is there someone whose parent is dino?". The answer is yes: dino is a parent of manuela.

Summing up, on the one hand logic programming and logic programs have the following advantages:

- **Declarative programming**: the paradigm allows for declarative programming, that is, programming through sentences which declare *what* to compute. The *how* is delegated to the underlying logic engine. Declarative programs are usually more concise and easier to understand than *procedural programs*, which are written through operational statements which declare directly *how* to compute.
- **Interactive programming**: the paradigm supports interactive programming. Users can write logic programs and then interact with them by means of queries (i.e. submission of goals).
- **Meta-programming**: many of the Prolog implementations support meta-programming. Given a logic program, it is possible to submit

[17] `http://www.swi-prolog.org/`
[18] `http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome`

queries which add or subtract clauses to/from the source code of the program as a side-effect.

Examples of these queries are `assert(Clause)`, which causes the addition of `Clause` to the program, and `retract(Clause)`, which cause the deletion of `Clause` from the program, if present.

On the other hand, they suffer from the following disadvantage:

- **Inefficiency**: despite several adopted optimizations, modern Prolog implementations are usually slower and thus less efficient than other programming languages or paradigms. This is mostly due to the declarative nature of logic programs and the proof-search strategy required by the resolution process. In most cases, searching for a proof in a solution space is much more computationally-expensive than straight-forwardly executing a procedural program.
- **Hard to debug**: logic programs are very hard to debug because of their lack of typing. Furthermore, it is very difficult to apply clean code rules to programs.

2.4.4 Logic-based smart contracts

The logic programming paradigm could be employed in the context of BCTs, rethinking smart contracts as logic-based entities [35]. For example, the EVM could be replaced with a *logic engine*, allowing end users to express smart contracts as logic programs. In this scenario, invocation transaction could be rethought as well to include a goal. This goal would be submitted to the target smart contract as a query to its source code.

On the one hand, end users would benefit from the declarative nature of logic programming, which could ease the development of smart contracts as well as the comprehension of their source code. Furthermore, logic programming would enhance the *inspectability* of smart contracts' source code. Since Prolog is an interpreted language, logic program could be directly deployed on the blockchain without the need of compilation and subsequently inspected seamlessly. Finally, meta-programming could also be employed as a means to change smart contracts' source code over time. This would allow for bug fixes and updates of the source code. More in detail, smart contracts could be endowed with both a static knowledge base (KB) containing immutable rules and facts, and a dynamic knowledge base containing rules and facts subject to change through meta-programming, allowing for a *controlled mutability* of their source code.

On the other hand, the adoption of logic programming would come with the cost of inefficiency, causing the execution of invocation transactions to be less performing.

Summing up, if system performance is not a critical requirement, the adoption of logic programming could be used as a viable alternative to existing approaches for smart contracts management, benefiting from all the advantages describe above.

Chapter 3

Vision

In this chapter, we focus on the concept of *proactive* smart contracts, paving the way towards the realization of a system meant to support their execution. In section 3.1 we discuss the idea behind this work, providing a description for proactive smart contracts and our goal. In section 3.2, we list the requirements of the system we wish to realize, along with a glossary in section 3.3 for clarity purposes. In section 3.4, we provide a set of scenarios in order to better understand the system dynamics, and we finally carry out a feasibility study in section 3.5, producing a first logic architecture of the system.

3.1 Idea

As explained in section 2.2, smart-contract-enabled BCTs allow for the creation and the deployment of smart contracts; end users can interact with them by publishing transactions and, in some BCTs, smart contracts can in turn synchronously send messages to other contracts.

Referring to Ethereum, let us observe what follows:

- **No proactivity**: smart contracts are essentially OOP objects. They are endowed with their own internal state and transactions issued to them can be conceived as *remote method invocations*. They encapsulate no control flow and the one that executes their code actually originates from the transaction issuer. In other words, smart contract are *passive* entities and the interaction with them is strictly *data-driven*, since they do nothing until a transaction is issued to them.

- **Messages $\neq$ Transactions**: messages differ from transactions since they are not provided with a digital signature and their interaction model is *always* synchronous; as depicted in Figure 3.1a, after sending a message smart contracts must wait for a response before proceeding with

execution. On the one hand, the lack of signatures is not a problem, since end users are not able to forge messages by design: messages exist only within the Ethereum system. On the other hand, the synchronous nature of messages prevents smart contracts from performing any kind of asynchronous interaction. As a result, smart contract are forced to wait for responses to their messages even when they are not interested in them.

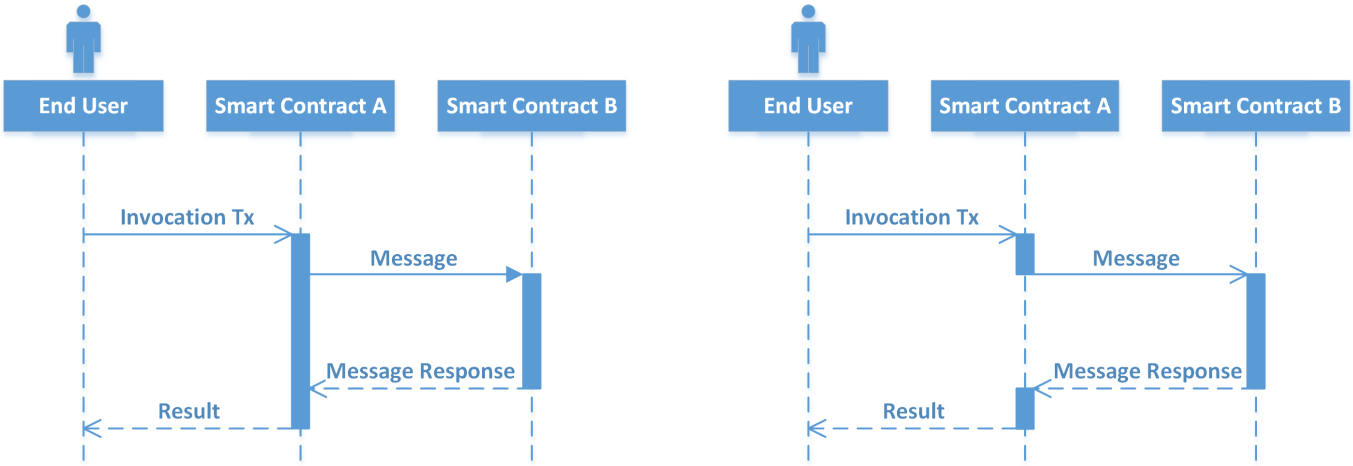
- **Irreversible deployment**: since smart contracts are deployed on the blockchain, their source code results immutable. This matter has two important consequences: code errors cannot be fixed/updated and smart contracts cannot be eliminated, since both operations would require to modify the block where their source code is stored, invalidating the blockchain integrity. This leads to the presence of buggy smart contracts on the blockchain which soon fall into disuse after their deployment.

Among these issues, the passive nature of smart contracts is the most limiting one. Because of it, smart contracts cannot perform periodic tasks, such as simple periodic payments in cryptocurrency-oriented BCTs, or react to events which occur within the system, such as the publication of transactions which are not addresses to them. More generally, they cannot play an active role in the execution of the terms of a physical contract.

Our goal is to overcome these problems. Thus, the core idea behind this work concerns both the study and the design of *proactive* smart contracts and the implementation of a BCT which supports them as a proof-of-concept. With respect to smart contracts, by the term "proactive" we refer to *(i)* the ability to perform computation even without external *stimuli*; *(ii)* the ability to interact with other smart contracts *asynchronously*, as depicted in Figure 3.1b; *(iii)* the ability to be reactive to time and to transaction issued to them. Furthermore, our aim is to allow end users of the system to modify the source code of smart contracts over time, in a secure and controlled way. In the next section, we formalize our idea in a detailed description of both functional and non-functional requirements which the system must fulfil.

3.2 Requirements

The system must be realized following the general architecture of modern BCTs; thus, it must consist of a set of nodes connected to each other in a peer-to-peer network and rely on a public or private blockchain to provide its functionalities in a distributed fashion. End users interact with the system by means of messages called *transactions* or *queries*. The former are used to



(a) Synchronous message

(b) Asynchronous message

Figure 3.1: Sequence diagrams depicting the possible interaction models of smart contracts' messages.

interact actively with the system, changing it's state; the latter are used only to retrieve information about the current state of the system.

3.2.1 Functional requirements

Nodes: the system nodes must continuously listen for incoming transactions. Each node must store a copy of the blockchain and participate in a consensus algorithm to update it consistently along with its peers. The blockchain must be used to store transactions published by end users. When a transaction is received, a node must relay it to its peers, spreading it over the network. Nodes are also in charge of executing transactions. As soon as a new block is committed to the blockchain, each node must execute all the transactions included in the block.

Furthermore, nodes must continuously listen for incoming queries. When a query is received, a node must reply with the desired information. Queries must allow end users to retrieve a list of all the existing smart contracts or retrieve the source code of a desired smart contract.

Smart contracts management: the system must act as an execution environment for smart contracts and enable end users to create, invoke, update and destroy them by means of transactions. More specifically:

- The system must support the execution of *four* different types of transaction, which end users are allowed to publish. These types are *(i) creation*

transactions, used to create a smart contract; they contain the source code of the smart contract and the initial operation that the smart contract must perform upon its creation (if any). They also specify whether the interaction with the newly smart contract should be *free* (i.e. everyone can invoke it) or *restricted* (i.e. it can be invoked by the creator only); *(ii) invocation* transactions, used to invoke a smart contract; they contain the operation to be invoked with respect to a desired smart contract; *(iii) update* transactions, used to update the source code of a smart contract; they include the *operation* to be either added or removed from the source code of the desired smart contract; *(iv) destruction* transactions, used to destroy a smart contract.

- The system must regulate the execution of smart contracts' operations in order to correctly manage non-terminating computations.
- The system must prevent the execution of non-deterministic code, in order avoid the creation of inconsistencies among nodes. Thus, the execution of smart contracts' operations must be strictly deterministic.

Smart Contracts: smart contracts can be created and destroyed by end users by means of creation transactions and destruction transactions respectively. Furthermore, they must:

- Encapsulate their own state and expose a set of *operations* which can be invoked through invocation transactions.
- Be able to perform an operation specified by the creator when they are started.
- Be reactive to transactions which are addressed to them.
- Be *reactive to time* and thus be able to support the *postponed* execution of operations as well as the *periodic* execution of operations.
- Be able to send proper invocation transactions *asynchronously* to other smart contracts.

3.2.2 Non-functional requirements

Accountability: for each transaction, the system must keep track of the entity, either end user or smart contract, that published it. This non-functional requirement is aimed at identifying system attackers or faulty entities.

Security: before being included into blocks, transactions must be validated. They must satisfy both the properties of *authenticity* and *integrity*; more precisely, for a given transaction the system must verify if the alleged issuer corresponds to the claimed one and if its content hasn't been altered. If a transaction does not fulfil this properties, it must be discarded.

Furthermore, the system must adhere to the following policy in order to properly regulate interactions with smart contracts: *(i)* every end user can create smart contracts; *(ii)* a smart contract can be updated and destroyed by its creator only; *(iii)* a smart contract can be invoked by its creator only if the interaction with it is restricted, or by everyone otherwise.

Finally, the system must prevent a transaction from being executed more than once.

3.3 Glossary

For the sake of clarity, in this section we provide a description for each term or phrase that could result ambiguous or too vague in this context.

Term/Phrase	**Meaning**
Entity	Either a smart contract or an end user.
Operation	A piece of smart contracts' code that can be object of computation, such as a function or a logic predicate. We use this term to abstract away from any programming paradigm.
Periodic operation	An operation which is executed periodically at a fixed time rate once invoked.
Delayed operation	An operation which is executed with a specific delay with respect to when invoked.
To *publish* a transaction	The act of sending a transaction to a node of the system.

3.4 Scenarios

Scenario 1 : an end user writes the source code of a smart contract that he/she wishes to create. The source code consists of a set of operations which can be invoked. Then, he/she packages the source code in a creation transaction, specifying an operation to be executed from the smart contract upon its creation. The transaction is issued to a node of the system. The node checks its validity (i.e. if the transaction satisfy both the properties of authenticity and integrity); if the transaction is valid, the node relays it to its peers. Otherwise, the transaction is discarded. In the former case, the transaction will be eventually included into a block and thus executed by each node, causing the creation of a new smart contract. When created, the smart contract will execute the operation specified by its creator.

Scenario 2 : an end user wishes to invoke an already created smart contract. First, he/she sends a query to one node of the system in order to retrieve and consult the source code of the desired smart contract. Then, he/she packages the operation that he/she wishes to invoke into an invocation message, sending it to a node. The transaction will be eventually included into a block if it is valid *and* if the issuer is allowed to interact with the smart contract. When the transaction is executed, the target smart contract executes the operation specified in the transaction.

Scenario 3 : an end user wishes to create a smart contract and make it executing an operation periodically, i.e, once every X seconds. First, he/she writes the source code of the smart contract including the periodic operation in it. Second, he sends a proper creation transaction to a node and waits for the smart contract to be created. Third, he packages the periodic operation into an invocation transaction for that smart contract and sends the transaction to a node. When the transaction will be executed, the smart contract will start to execute that operation periodically.

Scenario 4 : an end user wishes to update an already created smart contract. He/she packages the operation to be added or removed from the source code of the smart contract into an update transaction, sending it to a node. In this case, the transaction will be eventually executed if it is valid *and* if the issuer is the creator of the smart contract. The same logic applies to destruction transactions.

3.5 Problem analysis

In this section we study the problems arising from the requirements, carrying out a feasibility study in subsection 3.5.1; subsequently, we draw a first logic architecture of the system in subsection 3.5.2.

3.5.1 Feasibility study

In order to rely on a shared, replicated blockchain and update it consistently over time, nodes have to employ a consensus algorithm. As explained in subsection 2.1.2, the type of blockchain affects the choice of the consensus algorithm. Since there are no requirements regarding performance and scalability, the system could rely on either a public or private blockchain, adopting the most appropriate consensus algorithm. After this premise, we analyse the system requirements providing possible solutions for their fulfilment.

Node responses: end users interact with the system by means of messages called transactions and queries. This interaction is clearly of type *request-response*. End users send messages to nodes which replies accordingly. The requirements establish how nodes should reply to queries, being vague on how and when they should reply to transactions. As a result, nodes could reply to a transaction at different times, that is, *(i)* when the transaction is received, *(ii)* when the transaction is validated, or *(iii)* when the transaction is included into a block and thus executed.

Transaction security: The system must ensure both authenticity and integrity of transactions. This can be reached by following the approach adopted by other BCTs, that is, through asymmetric cryptography. End users must be provided with a pair of cryptographic keys and sign transactions with their own private key. Upon the receipt of a transaction, nodes can verify the signature with the public key of the issuer before sending it to their peers and including it into a block. Since smart contracts must be able to send transactions as well, it is necessary for them to be provided with a pair of keys too. Unfortunately this is not possible; we cover this matter later.

Identities: within the system, each entity must be endowed with a unique identity. Every transaction sent to nodes must contain the identity of the issuer for accountability purposes. The representation of identities is affected by the type of blockchain which will be employed by the system. Let us examine the two possible cases.

- A public blockchain is adopted. In this case, end users would generate their own identities. Thus, it would not be possible to rely on representations such as nicknames. This is because different end users could choose the same nickname as their identity, preventing the system to uniquely identifying them. For a given end user, his/her identity could be represented by his/her public key, its hash, or another unique identifier (such as UUIDs).
- A private blockchain is adopted. In this case, the system would include a CA in charge of dealing with identity management. End users' identities could be generated by either end users themselves or by the CA. In both cases, these identities would need to be enveloped in a certificate released by the CA to end users in order to be considered valid. In their transactions, end users would include both their identity and certificate. With this approach, the system would have total control on all the identities. Thus, end users could be identified through a nickname, a progressive number, a UUID, or through their public key.

In both cases, smart contracts' identities would be generated and controlled by the system. They could be created according to a fixed strategy. For example, the identity of a smart contract could be represented by a simple progressive number, a nickname, or be generated by combining the issuer's address with a progressive number and computing the hash such a combination. In case a private blockchain is adopted, smart contracts would have to be provided with a certificate for their identity in order to properly send transactions. Otherwise, end users could pretend to be a smart contract and forge a transaction without certificate, which would be considered valid.

Smart contracts: The system must allow for the update and destruction of smart contracts. As a consequence, smart contracts' source code cannot be stored on the blockchain since it would be immutable. Thus, it is required for the system to store source codes at the application level in order to modify or delete them. Furthermore, there are no restrictions on the programming language to adopt for the development of smart contracts. This means that they could be written either with a *compiled* or an *interpreted* programming language. In the first case, it would be necessary to follow an approach *à la Ethereum*, providing every node with a compiler for the chosen language and eventually a virtual machine for interoperability purposes. An update to the source code of a smart contract would require to re-compile it, leading to the loss of the smart contract's internal state. In the second case, it would be necessary to provide every node with an interpreter for the chosen language.

Updates to the source code would not require to compile it again, preventing state losses.

Non-determinism: The system must prevent smart contracts from executing non-deterministic code. Smart contracts can possibly execute non-deterministic code if the language employed for their writing supports non-deterministic computation. Thus, this issue could be solved by *(i)* creating (and adopting) a new language with no support for non-deterministic computation; *(ii)* modifying (and adopting) any programming language in such a way to remove support for non-deterministic computation; *(iii)* adopting any programming language with no restrictions; in this last case, nodes should inspect both creation and update transactions, invalidating them if any non-deterministic operation is found.

Non-terminating computation: The system must deal with infinite loops that could occur when a smart contract is executing some operation. To solve this problem, the system could follow the approach adopted by Ethereum, that is, employing a pricing model. Nonetheless, this approach would strictly require the presence of a native cryptocurrency. Another solution would be to allocate a limited amount of time T for the execution of each operation. In this case, for a given invocation transaction the nodes could either

- simulate the execution of the operation it contains before including it into a block. If the execution time exceeds the amount T, the transaction is marked as invalid and thus discarded.
- normally execute the operation contained in it when it is included into a block. If the execution time exceeds the amount T, nodes revert all the side effects caused by the execution.

Reactivity to time: smart contracts must be reactive to time. This means that they must be *time-aware* in the first place, that is, be provided with a notion of time. Since every smart contract is replicated over the system nodes, the presence of a shared, global time is crucial in order to avoid inconsistencies between a smart contract and its copies. For example, let us suppose what follows: an end user issues an invocation transaction to a smart contract, which cause it and all its copies to schedule the execution of an operation at a certain point in time. In this scenario, if nodes do not share the same time zone and the copies of the smart contracts relies on the node's local time, the scheduled operation would end up being executed at different times in different nodes, leading to inconsistencies of the global application state. Since the system

relies on a blockchain, blocks timestamps could be used as a solution to this problem. Smart contracts could use the timestamp of the most recent block as a shared and synchronized global time, which gets updated every time a new block is added to the chain.

Smart contracts' asynchronous interaction: smart contracts must be able to interact asynchronously with other contracts by means of proper transactions. As stated before in this section, transactions must be endowed with a signature in order to be validated properly. To adhere to this requirement it could be possible to:

- Provide smart contracts with cryptographic keys. In this case, they would sign transactions as end users do. Nevertheless, the creation of these keys represents a problem when performed within the system. This is because most cryptographic schemas rely on sources of non-determinism to generate keys. As explained in subsection 2.2.2, the execution of code must be strictly deterministic, making this approach invalid. As a workaround, the task of creating keys for a smart contract could be delegated to the end user that wishes to create it. This approach would be invalid too, because end users would have to include the key pair in their creation transaction, exposing the private key to everyone. Furthermore, smart contracts would not be able to securely store their private key, as their source code and storage are public.

- Rely on *transaction hash chains*. In this case, transactions sent by smart contracts would not be endowed with a signature but rather with a hash pointer to the transaction which caused their creation. Assuming that smart contracts send transactions only when invoked, this approach would cause every transaction issued by a smart contract to be a part of a tamper-proof hash chain in which the first transaction is always issued by an end user.

  To validate transaction with no signature, nodes would need to traverse its chain up to the the origin, that is, until a transaction issued by an end user is found. The validity of such transaction would assert the validity of the ones with no signature. Although this approach would allow smart contracts to be freed from cryptographic keys, it presents two major flows. First, smart contracts would not be allowed to send transactions on their own, since they would need to include an hash pointer to a transaction they do not have. Second, with this approach a malicious end user could pretend to be a smart contract by forging a transaction with no signature and setting its hash pointer to any already validated

transaction, forming a two-transaction chain. Since the first transaction of the chain is valid, the forged one would be considered valid as well.

- Provide nodes with cryptographic keys. In this case, nodes would be in charge of signing transactions which smart contracts wish to send. These transactions would then be validated in the same way as those sent by end users.

  Nonetheless, a malicious end user could pretend to be a smart contract and forge a transaction by signing it with his/her own private key. When receiving a forged transaction, nodes would not be able to verify if the signature has been made by a node or by a malicious end user. This issue could be solved by relying on a private blockchain, introducing a CA in the system. In such case, certificates issued by the CA would *(i)* assert the identity of the owner, *(ii)* assert the validity of a public key, and *(iii)* assert the nature of the owner (i.e. either node or end user). Following this approach, transactions would be endowed with both a signature and a certificate. For transactions issued by end users, the signature would be made by end users themselves. For transactions issued by smart contracts, the signature would be made by nodes. Transaction sent by a smart contract would be valid if endowed with a valid signature and a certificate which asserts that the signatory is a node. Note that this approach does not require to assign certificates to smart contracts.

Smart contracts' synchronous interaction: smart contracts are not able to interact *synchronously* with each others. This prevents smart contracts to wait for a response to a previously sent transaction when needed. A form of synchronous interaction could be implemented introducing a new type of transaction, namely, *acknowledgement* transactions, and using them as follows: if a smart contract A sends an invocation transaction to a smart contract B, then B executes the operation contained in the transaction and *asynchronously* sends an acknowledgement transaction back to A, containing the result of the operation.

3.5.2 Logic architecture

The most critical issue which arises from subsection 3.5.1 is the one related to signatures in smart contracts' transactions. The feasibility study showed that the only viable solution to this problem consists in relying on a private blockchain and on the presence of a CA, allowing nodes to properly sign smart contracts' transactions.

The need of both a CA and asymmetric cryptography for the validation of transactions makes it possible to define a logic architecture of the system in terms of structure, interaction end behaviour. The subjects involved in the system are:

- **End users**: they are external to the system and must be provided with a pair of cryptographic keys and a certificate issued by the CA in order to be allowed to interact with the system. End users are able to *(i)* create, invoke, update, and destroy smart contracts by means of transactions and *(ii)* retrieve information about smart contracts through queries.
- **Nodes**: they make up the system, forming a peer-to-peer network. Nodes are in charge of *(i)* managing the blockchain, *(ii)* providing an execution environment for smart contracts, and *(iii)* serving transactions and queries. Furthermore, they must be provided with a pair of cryptographic keys and a certificate issued by the CA to properly sign transactions on behalf of smart contracts.
- **Smart contracts**: they are deployed by end users and invoked by means of invocation transactions. Thus, they are reactive to transactions issued to them and reactive to time as well. They are able to *asynchronously* send transactions to other smart contracts.
- **Certification Authority (CA)**: it is a trusted component within the system which is in charge of dealing with identity management. More specifically, it releases certificates to end users and nodes. Given a certificate, it is possible to assert whether the owner is a end user or a node.

Figure 3.2 depicts the logic architecture of the system. End users and nodes must send a request to the CA in order to obtain a certificate. Upon the receipt of a request, the CA asserts the nature of its issuer, creates a proper certificate, and sends it back to the issuer. Thus, both end users and nodes communicate with the CA through *request-response* interactions.

Since nodes are linked in a peer-to-peer network, they relay transactions and blocks to their peers by means of *message passing* interactions too.

The requirements do not describe the interaction between nodes and smart contracts which occurs when an invocation transaction is received. They could communicate through *message passing* or *method calling*, depending respectively on the active or passive nature of smart contracts.

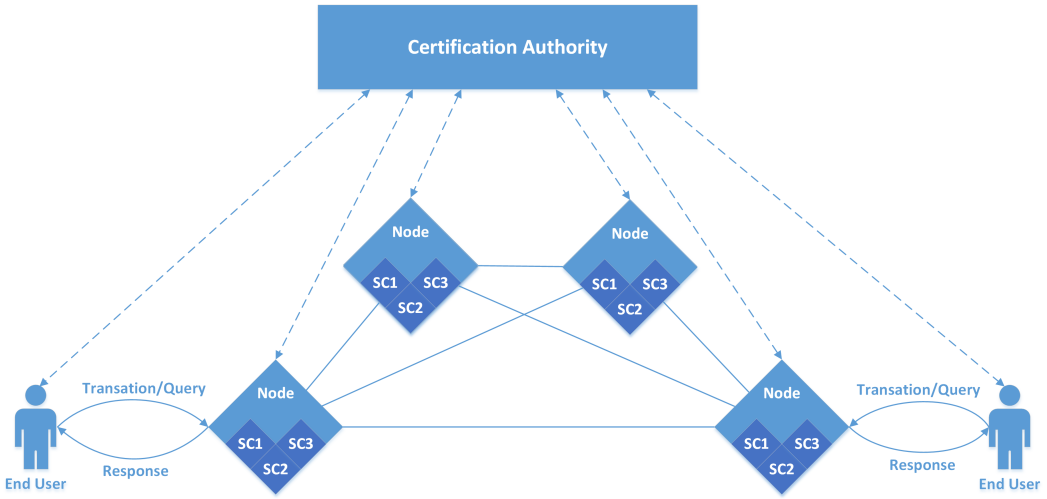


Figure 3.2: The logic architecture of the system. Nodes are linked in a peer to peer network. Every smart contract (SC) is replicated over the nodes. End users interact with the system by means of transactions and queries. Nodes and end users must request a certificate to the CA in order to be provided with a valid identity. In addition to smart contracts, each node stores a copy of the blockchain, which has not been included in the figure in order to preserve its readability.

Chapter 4

Design

After the previous chapter, where we established a set of requirements an analysed the deriving problems, we now focus on the system design. In section 4.1 we establish a general architecture of the system after discussing our design choices. In section 4.2 we describe in detail each component of such an architecture.

4.1 Architectural design

In this section we discuss our design choices, taken after the considerations carried out with the feasibility study in subsection 3.5.1. As a result, we subsequently refine the logic architecture of the system proposed in subsection 3.5.2, focusing on the interaction between components.

4.1.1 Design choices

During the feasibility study carried out in subsection 3.5.1, a couple of design choices were taken in advance to fulfil two system requirements. More precisely, asymmetric cryptography must be adopted in order to assert both authenticity and integrity of transactions and the introduction of a certification authority is necessary to make it possible for smart contracts to send proper invocation transactions. Furthermore, the security requirements implicitly split the transaction flow in different phases. When a transaction is received by a node, it enters the *validation* phase, which is aimed at verifying both its signature and certificate and validating it against the transaction policy. When this phase ends, the transaction gets discarded if it is not valid. Otherwise, the transaction enters the *broadcast* phase, where it is spread from the node to its peers. Eventually, the transaction will be included into a block.

At this point, the *execution* phase begin and the transaction is executed by all the peers.

With respect to the remaining requirements, the feasibility study showed different approaches which can be adopted to satisfy them. We now describe our design choices, providing a motivation for each on of them.

Identities: since a private blockchain is adopted and identities must be certified by the certification authority, their representation can be chosen at will. Among the representations listed in subsection 3.5.1, none of them offers particular advantages with respect to the others. Thus, we decide to represent identities as follows[1]:

- The identities of end users and nodes are represented by the cryptographic hash of their public key. Given an hash function $H(\cdot)$ and the public key K_{public}, the corresponding identity is represented by "$H(K_{public})$".
- Smart contract identities are represented by the letters *sc* followed by the cryptographic hash of the combination between the creator's identity and a progressive number. Given an hash function $H(\cdot)$, the creator's identity ID, and a progressive number n, the identity of a smart contract is represented by "$sc + H(ID + n)$".

Certificates and roles: the identities of end users and nodes must be certified by the CA through certificates. Furthermore, given a certificate, it must be possible for a node to determine if the owner is either a end user or a node, in order to detect forged transactions. To make this possible, we introduce the concept of *role* within the system. We establish two roles to be included into certificates, namely, "node" and "user". In order to obtain a certificate, end users and nodes send their public key to the CA, which replies with a signed certificate containing the issued public key, the related identity, and a role. Quite obviously, the CA assigns the role "node" to nodes and the role "user" to end users. Smart contracts do not need a certificate since they borrow the nodes' one when they wish to send a transaction.

Security enhancements: since roles are employed to detect forged transactions, we decide to exploit them to grant a finer management of interactions with smart contracts. More precisely, we adopt a RBAC policy in order to regulate smart contracts invocations. Creation transactions will no longer specify the kind of interaction with respect to the newly smart contract (i.e. *free* or

[1]The symbol "+" is the concatenation operator between strings and the formalism $H(X)$ denotes the cryptographic hash of X obtained through the hash function $H(\cdot)$.

restricted), but rather a role, which establishes who is allowed to invoke it. If the specified role is "user", only end users will be allowed to invoke the smart contract. Similarly, if the specified role is "node", only other smart contracts will be allowed to invoke the smart contract. In order to preserve the "all-or-nothing" policy of the free/restricted approach, we introduce two dummy roles, namely, *root* and *any*. If a creation transaction includes the former, no one will be allowed to invoke the newly smart contract except for its creator. If a creation transaction includes the latter, everyone will be allowed to invoke the newly smart contract. Furthermore, the root role can be used for the creation of proper certificates for system administrators, granting them full access to the system. Thus, we redefine the overall interaction as follows: *(i)* every transaction must be endowed with a valid certificate in order to be accepted and processed by nodes; *(ii)* everyone can create smart contracts; *(iii)* a smart contract can be updated and destroy by its creator *and* by system administrators; *(iv)* invocations of a certain smart contract are regulated by the role specified in the creation transaction which caused its creation; regardless of the specified role, a smart contract can always be invoked by its creator *and* by system administrators.

Smart contracts nature: smart contracts could be modelled either as active or passive components. In the former case, they would encapsulate a control flow and nodes would interact with them through message passing. In the latter case, nodes would interact with them through method calling. Let us observe what follows: smart contract are perceived by end users as proactive, independent entities which can be created, invoked, modified, and destroyed through transactions. Nonetheless, interactions between end users and smart contracts are always mediated by nodes, and since nodes must be able to manipulate smart contracts having total control on them, we decide to model smart contracts as passive entities.

"Synchronous" interaction: as stated in the feasibility study, the lack of synchronous interaction can be a problem in several scenarios. For this reason, we decide to introduce *acknowledgement* transactions in order to simulate synchronous interactions. These transactions are meant to be sent by a smart contract A to a smart contract B after A is invoked by B through an invocation transaction. More specifically, when A executes an operation invoked by B producing a result R, it creates and send an acknowledgement transaction to B containing *(i)* the invocation transaction which B sent to A and *(ii)* the result R.

Infinite computation management: in order to correctly deal with infinite computations, the feasibility study showed that it is possible to either rely on a pricing model or impose an execution time limit for each operation. We decide to adopt the latter approach, since the former would require the implementation of a cryptocurrency, which is not in the scope of this thesis. The chosen approach offers in turn two possible solutions to the problem. The first consists in simulating the execution of operations before including invocation transactions into blocks, invalidating them if the simulations time exceed a certain threshold T; the second avoids this control, executing invocation transactions normally when they are included into a block, and reverting the side effects of an operation if its execution time exceeds T. We decide to rely on the first solution, since it prevents invalid invocation transactions to be included into blocks.

As a result, an invocation transaction is considered to be valid if three conditions are met. Firstly, it must satisfy both the properties of authenticity and integrity. Secondly, it must adhere to the transaction policy. Finally, the simulation of the operation which is contained in it must take less time than a certain threshold T.

Language for smart contracts: referring to the language to be adopted for the writing of smart contracts, we choose to rely on an interpreted and logic-based programming language. This is because *(i)* the interpreted nature of the language prevents state losses of smart contracts when update transactions are executed; it also enhances the *inspectability* of source codes, since they do not need to be compiled down to bytecode or binary [35]; *(ii) meta-programming* can be used to ease updates to smart contracts' source code; *(iii)* the declarative nature of programs written in this language could ease both their development and their understanding; *(iv)* smart contract can be structured in such a way to modify their behaviour in a controllable way, as better described in subsection 2.4.4. As a result, the operations which make up the source code of a smart contract are either facts or rules. Operations sent in invocations transaction are goals, which can be conceived as query to be submitted to the target smart contract.

Preventing non-deterministic computation: the feasibility study showed different approaches to prevent the execution of non-deterministic code. We discard the idea of writing a brand new language with no support for non-deterministic computation, since its realization would go far beyond the scope of this thesis. Furthermore, we discard the approach which consists in inspecting source codes and invocation transactions, since it can be avoided by adopting a modified version of a programming language with no support for

non-deterministic computations.

Summing up, smart contract are modelled as logic entities written in a language with no support for non-deterministic computation. The choice of this language is delegated to the implementation phase.

Node responses: when a transaction is received, nodes could reply to end users at different times with different responses or not reply at all. We decide to model nodes so that they can reply at three different times, that is, *(i)* when the transaction is received, *(ii)* when it is validated, or *(iii)* when it is included into a block. As a result, end users can send transactions in three different modes, namely, *async*, *sync*, and *commit*. Depending on the chosen mode, they will receive a response either when the transaction is received by the node, when the node ends the validation process, or when the transaction is successfully included into a block and thus executed.

Node splitting: the feasibility study showed that source codes of smart contracts cannot be stored on the blockchain, otherwise they could not be updated or deleted. Thus, they must be stored at "application level". In general, modelling each node as a monolithic module which provides all the functionalities is considered as bad practice. This module would probably lack flexibility and would be difficult to both understand and modify. With that being said, we decide to split each system node into two modules, namely *core* and *logic interpreter*. The former is in charge of dealing with blockchain management and peer-to-peer connectivity. The latter is in charge of managing smart contracts and their execution, acting as a logic interpreter.

Multicast/Broadcast: each invocation transaction is addressed to a single smart contract (i.e. *unicast*). In many real-case scenarios, it would be useful for end users and smart contracts to send the same transaction to a sub-set of smart contracts or to all of them. To support this extra feature, we decide to introduce both a *multicast* and *broadcast* mode for invocation transactions. Given one of these transactions, it is invalid if *(i)* it causes at least one timeout or *(ii)* the issuer can invoke none of the recipient smart contracts.

4.1.2 General architecture

In subsection 3.5.2 we provided a first architecture of the system. However, we think that it is necessary to revisit this architecture as it presents two issues, related to both security and end users' interaction with the system.

The security issue concerns the certification authority. Within the system, it is the trusted entity which is in charge of certifying the identities of all

end users and nodes. Being the only one, it represents a *single point-of-trust*. If an attacker manages to gain control over it, the trust model of the whole system would be compromised. In order to deal with this problem, we replace the single certification authority with a hierarchy of certification authorities. In this hierarchy, CAs are arranged at levels, forming a tree. The first level includes a single CA, called *root*, which is responsible for releasing a certificate to each sub-CA at the second level. These ones are in turn responsible for releasing a certificate to each sub-CA at the third level, and so on. The last level includes the CAs which are responsible for releasing certificates to end users and nodes, called *leaves*. By design, there is no limit to the number of levels that a real deployment of the system could employ. To check the validity of a certificate X released to either a node or an end user it is necessary to: *(i)* check if the certificate signature is valid; *(ii)* contact the CA who released it and require its certificate; *(iii)* repeat these two steps until the root CA is reached; if its certificate is valid, then the certificate X is valid too.

While adopting such a hierarchy, if an attacker were to gain over a sub-CA, he/she would invalidate only a subset of the released certificates. Nonetheless, the root CA still represents a *single point-of-trust*. Thus, this approach is used to mitigate the problem and not to entirely solve it.

The other issue is as follows: actually, end users have no support to interact with the system. The requirements do not include the realization of a module meant to mediating end users' interaction with the system, in order to ease both the creation and sending of transactions and certificate requests. We introduce a new type of system node, namely, *client* nodes. Clients consist of a web application which allows to *(i)* request a certificate to a CA; *(ii)* create and send both transactions and queries. Each client is meant to mediate the interaction with the system on behalf of a single end user.

Summing up, the system now includes a hierarchy of CAs and two types of node, namely, *full* nodes and *client* nodes. These refinements are depicted in Figure 4.1, which represents the general architecture of the system.

4.1.3 Interaction

The systems is made up by several heterogeneous and distributed components. In order to properly interact with each others, the components need an interaction media meant to support their distributed nature, namely, the Internet. Thus, all interactions will occur over the Internet and will be regulated by different protocols depending on the interacting parts.

We decide to employ the HTTP protocol for the regulation of *request-response* interactions. Both clients and full nodes request certificates to sub-CAs by means of HTTP requests, and both transactions and queries are sent

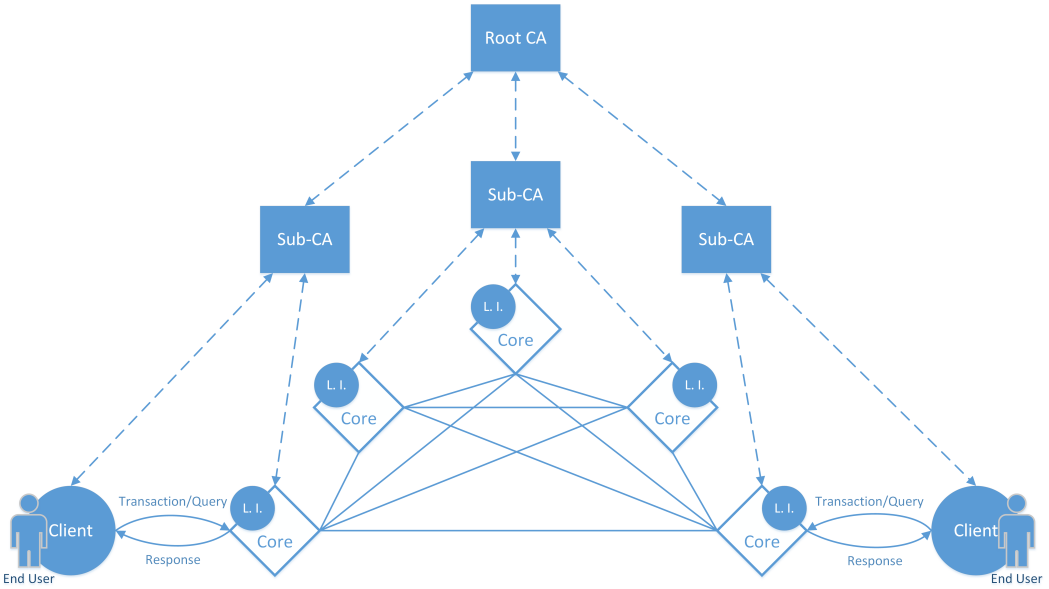


Figure 4.1: The general architecture of the system, a refinement of the one depicted in Figure 3.2. In this representation, the CA hierarchy has two levels. Sub-CAs shall request a certificate to the root-CA, while end users and full nodes shall request a certificate to one of the sub-CAs. Each full node is divided in two modules, namely "core" and "logic interpreter" (L.I.). The blockchain (stored by each core) and smart contracts (managed by each logic interpreter) are not depicted in order to preserve readability. The interaction between end users and nodes is mediated by clients.

by clients through HTTP requests as well. As a result, CAs and full nodes expose a web server in order to correctly manage these requests. Within the system context, there are seven different HTTP requests:

- `GenerateCertificate`: GET request which can be sent to CAs in order to request and obtain a new certificate. The relative URL path is `/generateCertificate`. These requests must contain a public key and a role as query parameters, by the fields "publicKey" and "role" respectively.
- `RetrieveCertificate`: GET request which can be sent to CAs in order to retrieve their certificate. The relative URL path is `/getCertificate`.
- `SendTransactionAsync`: GET request which can be sent to full nodes in order to publish a transaction and get a response as soon as the transaction is received. The relative URL path is `/broadcastTxAsync`. These requests must contain the transaction to be published as a query parameter, by the field "tx".

- `SendTransactionSync`: request similar to the previous one. When receiving such a request, full nodes will reply only when the transactions is validated. The relative URL path is `/broadcastTxSync`.

- `SendTransactionCommit`: request similar to the previous one. Upon the receipt of such a request, full nodes will reply either when the transaction is validated (if it is not valid) or when the transactions is successfully included into a block (if it is valid). The relative URL path is `/broadcastTxCommit`.

- `Query`: GET request which can be sent to full nodes in order to retrieve information about the application state. The relative URL path is `/query`. These requests must contain the field "data" as a query parameter. The corresponding value can be either "smartContractIdentities" to retrieve the list of smart contracts' identities or the identity of a smart contract in order to retrieve its source code.

- `RetrieveBlocks`: GET request which can be sent to full nodes in order to retrieve a set of contiguous blocks. The relative URL path is `/blockchain`. Since the position of blocks is identified by their *heights*, as described in subsection 4.2.1, these requests must specify both a minimum and a maximum height as query parameters, by the fields "minHeight" and "maxHeight" respectively.

The protocol used to regulate full nodes inter-communication is strictly dependent on the consensus algorithm which will be employed. Since the system relies on a private blockchain, we are interested in "classical" consensus algorithm, both BFT and non-BFT, as explained in subsection 2.1.2. The choice of this algorithm, and thus of the interaction protocol, is delegated to the implementation phase, in chapter 5. In this phase we can simply assume that a consensus protocol is engaged by full nodes and the total ordering of transactions is thus ensured.

4.2 Detailed design

In this section, we first give a shape to blockchain blocks, establishing their structure and fields. Then, we set a fixed representation for both transactions and certificates. Finally, we focus on smart contracts and the main system components, giving a detailed description regarding both their behaviour and interaction.

4.2.1 Blockchain

The blockchain stores blocks at *heights*, with one block at each height. Each block is made up of a header and a body. The header contains meta information about the block, that is, *(i)* its height, *(ii)* the Unix timestamp of its creation *(iii)* the Merkle root of the application state reached after the execution of the transactions included into the previous block, *(iv)* the Merkle root of the transactions contained in the body, and *(v)* a hash pointer to the previous block in the chain. The body consists of a set of transactions.

Each time a new block is added to the blockchain, each full node synchronizes its local time with the timestamp of the new block. As a result, all the full nodes share the same (global) time, which is kept synchronized among them thanks to blocks timestamps. This notion of global time is essential to make smart contracts *time-aware*, and thus reactive to time.

The first Merkle root is included into headers to securely store a representation of the application state after the execution of some transactions. Clients can retrieve the data that makes up the application state, compute the corresponding Merkle root, and rely on the first Merkle root in the header of the most recent block to verify if the retrieved data is valid. The second Merkle root is included into headers in order to guarantee *proof-of-membership* for transactions.

All the blocks must adhere to this structure and contain at least all the described fields. Additional fields could eventually be added to both header and body if required by the consensus algorithm which will be employed.

4.2.2 Transactions and certificates

Given the adoption of logic programming for smart contracts, we decide to model transactions and certificates as logic terms. Furthermore, we decide to split certificates in two categories, namely, *intermediate* and *entity-or-node* (EON) certificates. The former are released by a non-leaf CA to another CA; the latter are released by a leaf CA to either a end user or a node.

Intermediate certificates consist of *(i)* the `PUBLIC_KEY` of the CA, *(ii)* an `EXPIRATION_DATE`, *(iii)* the `URI` of the CA which issued the certificate, and *(iv)* a `SIGNATURE` made by the CA which issued the certificate. What follows is the representation of an intermediate certificate, structured as a logic term.

```
certificate(PUBLIC_KEY, EXPIRATION_DATE, URI, SIGNATURE)
```

EON certificates consist of *(i)* the `IDENTITY` of the requester, generated by the public key, *(ii)* the specified `PUBLIC_KEY`, *(iii)* the specified `ROLE`, *(iv)* an `EXPIRATION_DATE`, *(v)* the `URI` of the CA, and *(vi)* a `SIGNATURE`, made by the

CA with its private key. They are represented as:

```
certificate(IDENTITY, PUBLIC_KEY, ROLE, EXPIRATION_DATE, URI, SIGNATURE)
```

Every certificate must contain the URI of the CA which released it in order to make the validation of certificates possible, as explained in subsection 4.2.6. Also, providing each certificate with an expiration date is usually a good practice to enhance security, as it forces certificate owners to request a new one periodically.

All transactions adhere to a fixed representation. As a result, full nodes can deduct their type only by inspecting their content. Transactions contain *(i)* the sender `IDENTITY`, *(ii)* a `BODY`, *(iii)* a `NONCE`, *(iv)* a `CERTIFICATE`, and *(v)* a `SIGNATURE`. The fixed representation is as follows:

```
transaction(IDENTITY, BODY, NONCE, CERTIFICATE, SIGNATURE)
```

`NONCE` is an integer value which represents the number of transactions which the sender has previously sent. Nonces are included into transaction in order to detect duplicates and for the creation of smart contracts' identities when necessary. The `BODY` establishes the transaction type and it can be one of the following terms:

- `create(theory([CLAUSE | OTHER_CLAUSES]), ROLE, INIT_OP)`: structured term used to create a new smart contract. The source code is represented by the list of logic clauses included in the first term; a logic clause can be either a fact or a rule. The second term, i.e. `ROLE`, defines who can invoke the newly smart contract, and can be either *any*, *user*, *node*, or *root*. The last term contains the initial operation, i.e. the goal, the must be submitted to the newly smart contract. A transaction with this body is a creation transaction.

- `assert(SC_ID, CLAUSE)`: structured term used to update the smart contract with identity `SC_ID`. It causes the addition of the logic clause `CLAUSE` to its source code. A transaction with this body is an update transaction.

- `retract(SC_ID, CLAUSE)`: structured term used to update the smart contract with identity `SC_ID`. It causes the removal of the logic clause `CLAUSE` from its source code, if present. A transaction with this body is an update transaction.

- `send([SC_ID1, SC_ID2 | OTHER_SC_IDS], GOAL)`: structured term used to invoke an arbitrary-long list of smart contracts. The first term is a list of smart contract identities. The second one is the `GOAL` that must

be submitted as a query to each smart contract. A transaction with this body is an invocation transaction, in unicast mode if the list contains one identifier, in multicast mode otherwise.

- `send(_, GOAL)`[2]: structured term used to invoke *all* the existing smart contracts, submitting the goal `GOAL` to each one of them. A transaction with this body is an invocation transaction, in broadcast mode.

- `ack(SC_ID, TRANSACTION, RESULT)`: structured term used to acknowledge an invocation transaction. The first term is the identity of the smart contract which issued the invocation transaction. The second term is the invocation transaction itself. The third term is the result computed by "executing" the goal inside the invocation transaction. A transaction with this body is an acknowledgement transaction.

- `destroy(SC_ID)`: structured term used to destroy the smart contract with identity SC_ID. A transaction with this body is a destruction transaction.

We decide to rely on the RSA algorithm for the generation of keys and validation of signatures, adopting 2048-bit long keys. More precisely, each signature is computed through the "RSAwithSHA256" algorithm. Given some data that needs to be signed, the algorithm first computes the SHA256 digest of it and then runs RSA to create a signature. Thus, given the hash function *SHA256(·)* and the RSA signature algorithm *SIGN(·)*, the `SIGNATURE` field of both certificates and transactions is calculated by *(i)* concatenating all the other fields, forming the string X, *(ii)* computing the SHA256 digest of X, formally *SHA256(X)*, and *(iii)* running the RSA signature algorithm with the digest as input, formally *SIGN(SHA256(X),* $K_{private}$*)*, where $K_{private}$ is the private key of either the certificate issuer or the transaction creator.

4.2.3 Smart contracts

Smart contracts are designed as logic programs. Every smart contract is endowed with a static knowledge base (KB) and a dynamic KB. The static KB stores immutable rules and facts, while the dynamic KB stores mutable ones. This neat division allows for a controllable mutability of smart contracts. "Library" facts and rules which are essential to smart contracts' functioning are stored permanently in the static KB, preventing their deletion. The source code is stored in the dynamic KB, making its modification possible.

[2]The underscore character ("_") is called an *anonymous variable* and unifies with any term.

Operations sent within invocation transactions can be conceived as goals which must be queried to the source code of smart contracts. In order to allow developers to define how their smart contracts should react to invocation transactions, we designed smart contracts' behaviour as follows: when receiving an invocation transaction containing the goal `Goal`, a goal-oriented computation `Body` is triggered if the source code of the target smart contract contains a rule such as `receive(Goal) :- Body`. Otherwise, the invocation fails. This approach allows developers to define an *interface* for their smart contracts, meant to group the goal-oriented computations which end users are allowed to trigger. Let us explain this concept with a trivial example. Listing 4.1 contains the source code of a smart contract, which stores data in a key-value fashion. Data can be read or written only through two apposite rules. If a end user sends an invocation transaction containing the goal `set_data(key3, data3)`, the smart contract is invoked with goal `receive(set_data(key3, data3))`, which causes the addition of a new key-value pair to the source code. Similarly, if the goal is `get_data(key1, Value)`, the smart contract is invoked with goal `receive(get_data(key1, Value))`, giving the solution `get_data(key1, value1)`. If the goal is `key_value(key1, Value)` the invocation fails, since there is no rule in the source code whose head matches with `receive(key_value(key1, Value))`.

```
% Facts.
key_value(key1, value1).
key_value(key2, value2).

% Rules which make up the smart contract interface.
receive(set_data(Key, Value)) :- assert(key_value(Key, Value)).
receive(get_data(Key, Value)) :- key_value(Key, Value).
```

Listing 4.1: An example of source code of a smart contract.

Finally, when created, a smart contract is endowed by default with a set of library terms, which allows it to *(i)* store the identity of its creator, *(ii)* consult the current global time, *(iii)* send invocation and acknowledgement transactions, and *(iv)* perform delayed and periodic computation.

4.2.4 Full nodes

Full nodes are composed of two separated modules, namely, *core* and *logic interpreter*. The former listens for transactions from both clients and smart contracts and relays them to other full nodes; it also participates in the consensus algorithm along with the other cores to establish a total ordering of transactions, storing them in the blockchain. The latter is in charge of *(i)*

validating and executing transactions, *(ii)* managing both the creation and destruction of smart contracts, *(iii)* running a *logic engine*, used to submit queries to smart contracts, aimed at updating or invoking them, *(iv)* building responses to transactions or queries to be sent back to clients. The application state of a full node is handled by the logic interpreter. At a give point in time, the state is represented by the set of smart contracts that currently exist.

Note that clients never interact with logic interpreters directly. This interaction is always mediated by cores, as described in the following section. This approach is adopted to keep the application state consistent among full nodes.

Interaction & behaviour

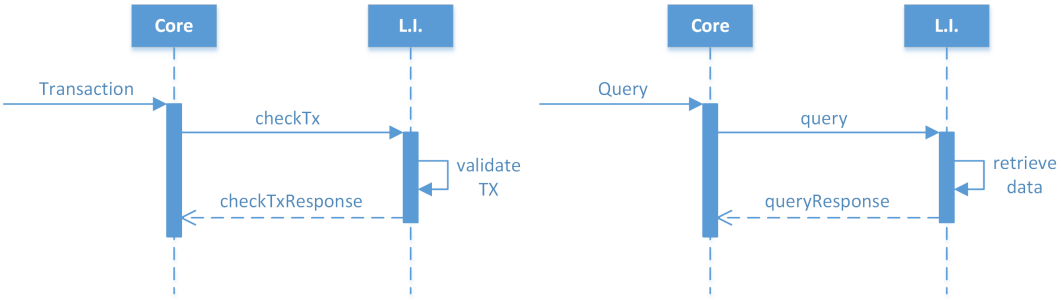
The interaction between the two modules is strongly inspired to the one adopted by Tendermint. They communicate through a series of messages, in a *request-response* fashion. A communication act is initialized from the core to the logic interpreter when it receives either a transaction or a query and when a new block is added to the blockchain. When the core receives a transaction through one of the three GET requests or by one of its peers, the interaction proceeds as follows:

1. The core packages the transaction in a `checkTx` message, and sends it to the logic interpreter. Then, it waits for a `checkTxResponse` message.
2. The transaction is received by the logic interpreter and enters the validation phase. The algorithm for checking if a transaction is valid is as follows.
    2.1. Check if the transaction is well formed, i.e. it adheres to the representation described in subsection 4.2.2.
    2.2. Check if its certificate is valid. The validation process of certificates is described in subsection 4.2.6.
    2.3. Check if the signature is valid using public key contained within the certificate.
    2.4. Check if the transaction adheres to the transaction policy. More precisely, for invocation, update, and destruction transactions, check if the sender is allowed to perform the operation contained within the transaction.
    2.5. Simulate the transaction and assure that its execution time does *not* exceed a predefined threshold T. During this steps, no changes are committed to the application state.

If any of these steps fails, the logic interpreter sends a negative `checkTxResponse` message back to the core. Otherwise, it sends a positive `checkTxResponse` message.

3. If the core receives a positive validation response message, it spreads the transaction to the other full nodes and stores it until it is included into a block. Otherwise, the transaction is discarded.

When a query is received, the core sends a `query` message to the logic interpreter, which replies with a `queryResponse` containing the desired information. Figure 4.2 depicts the interaction between core and logic interpreter when the former receives either a transaction or a query.



(a) The interaction flow between the parts when a transaction is received.

(b) The interaction flow between the parts when a query is received.

Figure 4.2: Sequence diagrams depicting the interaction between core and logic interpreter (L.I.) when either a transaction or a query is received by the core.

Whenever a block is created and added to the blockchain, the interaction proceeds as follows:

1. The core creates a `beginBlock` message and sends it to the logic interpreter. This messages contains meta information about the block, as its timestamp and height. Then, it waits for a `beginBlockResponse` message.

2. The logic interpreter synchronize its global time with the timestamp of the block. Then, it sends back a `beginBlockResponse`.

3. At this point, the core packages each transaction contained in the block in a `deliverTx` message. Then, it sends these messages to the logic interpreter, one by one. After sending one of them, it waits for a `deliverTxResponse` message before sending the following in the list.

4. When receiving a `deliverTx` message, the logic interpreter executes the transaction contained in it. Then, it builds and sends back a `deliverTxResponse` message, whose content depends on the type of transaction executed. For creation transactions, it contains the identity of the newly smart contract; for invocation transactions, it contains the invocation result (i.e. the *mgu* between the submitted goal and the source code of the target smart contract). Otherwise, the content is left empty.

5. After sending all the `deliverTx` message, the core sends an `endBlock` message in order to tell the logic interpreter that all the transactions within the new block have been sent. The logic interpreter replies with a `endBlockResponse` message.

6. Finally, the core a sends a `commit` message to the logic interpreter, meant to require the Merkle root of the new application state. It will be included in the header of the next block.

7. The logic interpreter computes the Merkle root and sends it back to the core inside a `commitResponse` message.

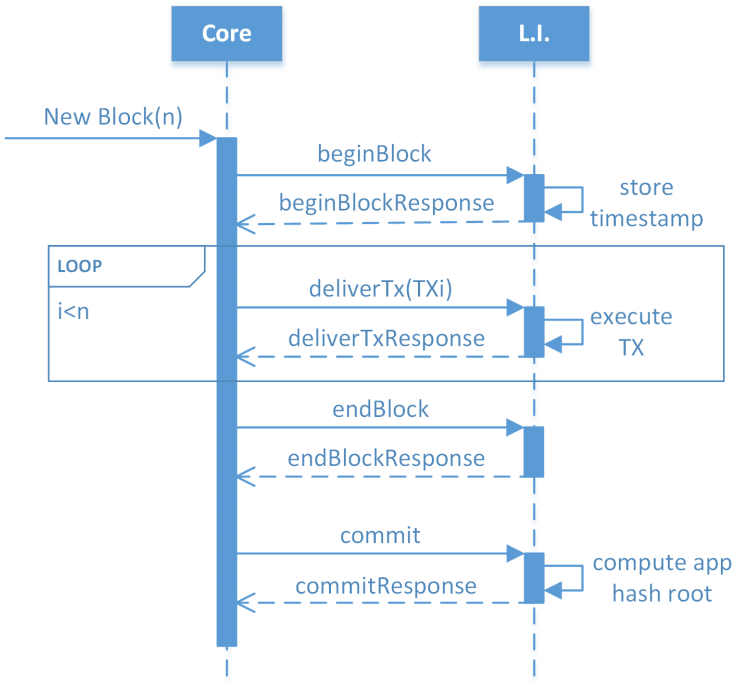


Figure 4.3: Sequence diagram depicting the interaction between core and logic interpreter (L.I.) when a new block is committed to the blockchain.

Logic interpreter

A logic interpreter is made up by several components, each one of them providing a different functionality. These components are orchestrated by a *Controller*, which encapsulates the overall application logic, making the logic interpreter behave as described previously in this section. The interaction occurs through *method-invocation*. The components are as follows:

- **Server**: component in charge of listening for messages sent by the core at port `5724`. As soon as a message is received, this component notifies the *Controller*.

- **Smart Contract Manager**: component in charge of storing smart contracts created by end users. It runs a *logic engine* to submit goals contained in invocation and update transactions to smart contracts. Thus, it is the one which effectively *(i)* simulate transactions during the validation phase and *(ii)* executes them when necessary, changing the application state. In order to properly simulate transactions without changing the application state, this component stores a copy of each existing smart contract. Each one of this copies is a *dummy* version of the original one, meaning that it has the same source code but is incapable of sending transactions or perform delayed/periodic tasks. Thus, given an invocation or update transaction issued to a certain smart contract X, its simulation is performed on the *dummy* copy of X, while its real execution is performed on X.

  Finally, every time that all the transactions in a block are effectively executed and a `commit` message is received, all the *dummy* copies are deleted and re-computed. This is done to preserve consistency between smart contracts and their *dummy* copies over time.

- **Http Client**: a simple http client meant to allow smart contracts to send proper transactions to the underlying core. More precisely, it is able to send `SendTransactionAsync` requests on behalf of smart contracts. Moreover, it makes it possible to *(i)* send `GenerateCertificate` requests to leaves CA in order to obtain a valid certificate for the full node, *(ii)* send `RetrieveCertificate` requests to any CA in order to retrieve its certificate.

- **Authorization Manager**: for each smart contract, this components keeps track of the roles which are allowed to invoke it. For a given transaction, this component is able to check whether the issuer is allowed to perform the operation specified in it.

- **Key Manager**: component in charge of dealing with cryptography operations within the system. It stores the key pair of the full node and is able to both create and verify RSA signatures, through the algorithm "SHA256withRSA". The key pair is generated at runtime by this component.

- **Time Manager**: component in charge of storing the system global time, making it available to the other components.

- **Certificate Manager**: component in charge of managing the full node's certificate, making it available to the other components when required. It requests a new certificate through the Http Client both when *(i)* the logic interpreter is started and *(ii)* the current certificate expires.

- **Transaction Validator**: component meant to carry out all the necessary controls on transactions in order to assert their validity. For each identity, it keeps track of the corresponding "expected nonce", that is, the nonce that needs to be included in the next transaction in order for that transaction to be valid. When invoked for the validation of a transaction, it in turn invokes *(i)* the *Authorization Manager* for verifying the transaction adherence to the access control policy, *(ii)* the *Key Manager* for verifying signatures, *(iii)* the *Http Client* for requesting certificates to be verified, and *(iv)* the *Smart Contract Manager* for simulating the execution of the transaction.

- **Transaction Builder**: component in charge of building proper transactions when requested by smart contracts. More precisely, smart contracts can invoke this component providing a `BODY` and their `NONCE` to obtain a well-formed transaction endowed with the full node's certificate and a proper signature.

Whenever a message is received by the Server, the Controller is invoked, which in turn invokes other components depending on the message type. For `checkTx` messages, the Transaction Validator is invoked to carry out the validation process on the corresponding transactions; for `beginBlock` messages, the Time Manager is invoked in order to synchronized the global time with the one contained in the messages; for `deliverTx` messages, the Smart Contract Manager is invoked for executing the corresponding transactions; for `commit` messages, the Smart Contract Manager is invoked to retrieve all the existing smart contracts and compute the Merkle root of the application state. The Merkle root is computed from a tree in which the source codes of smart contracts make up its leaves.

Core

Like logic interpreters, cores are made up by several components. First of all, the business logic is handled by *controller*, which is notified by the other components when some event occurs (i.e. a transaction is received, the logic interpreted replied to a previously sent message, ...). The other components are as follows:

- **Web Server**: component meant to listen for transactions and queries at port `5521`. Recall that transactions can be sent by both end users and smart contracts, through clients and logic interpreters respectively, while queries can be sent by clients only. The web server accepts five types of HTTP requests, namely, `SendTransactionAsync`, `SendTransactionSync`, `SendTransactionCommit`, `Query`, and `RetrieveBlocks`.
- **Peer-to-peer module**: component meant to manage communication with other full nodes; more precisely, its duty is to relay both transactions and blocks to other full nodes when necessary; transactions are relayed after being validated by the logic interpreter; a block is relayed when required by the consensus protocol which will be employed.
- **Blockchain Manager**: component meant to store the blockchain and attach new blocks to it when necessary.
- **Client**: component meant to manage the full node's inter-communication with the logic interpreter.

The consensus algorithm which will be adopted may require the addition of components outside from this list, in order to manage, for example, the creation of new blocks. These components cannot be modelled here, since the consensus algorithm has not been decided yet. Because of this, the description of both interaction and behaviour of the listed components is delegated to the implementation phase, where it will be possible to draw up a more accurate and complete list.

4.2.5 Clients

Clients mediates the interaction between end users and full nodes. Each client is essentially a web application which is in charge of managing the key pair and the certificate of a certain end user, enabling him/her to send both transactions and queries to a full node and inspect the blockchain. They do not store the blockchain nor do they participate in the consensus algorithm

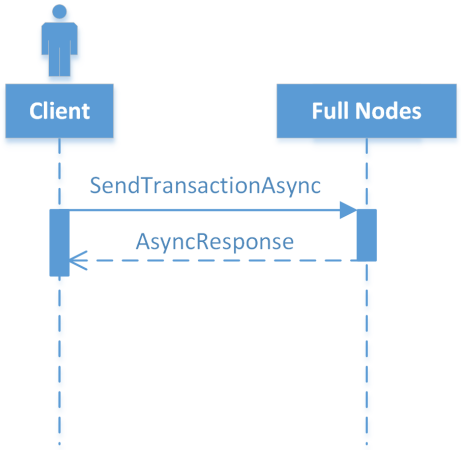
employed by full nodes. More precisely, clients are meant to ease the aforementioned interaction as much as possible. Thus, they transparently *(i)* generate the end user's key pair, *(ii)* request a new certificate to a leaf CA when necessary, *(iii)* store and update the progressive number that needs to be included in each transaction that the end user wishes to send (i.e. the `NONCE` field), and *(iv)* sign each transaction.

Clients can send transactions to full nodes through one of the three possible HTTP requests described in subsection 4.1.3. The content of full nodes' responses depends on the corresponding request, as depicted in Figure 4.4. Clients can also send `GenerateCertificate` requests to leaf CAs in order to obtain a new certificate for the end user. Clients can send `Query` requests to full nodes in order to retrieve information about the identities of smart contracts and their source code. Finally, clients can send `RetrieveBlocks` requests to full nodes to retrieve a certain set of contiguous blocks.

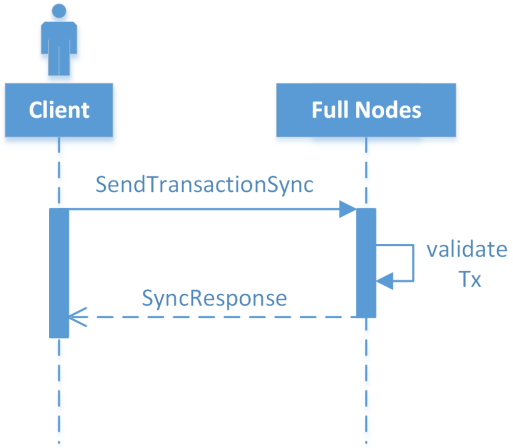
Since clients are web applications, end users interact with them through their web server. A end user can initialize the interaction with his/her (local) client through a GET request with path "/" at port `7098`. When receiving this request, the client replies with an HTML page, namely, the *HomePage*. From now on, the interaction proceeds through further GET requests, allowing the end user to carry out all the aforementioned operations. The HTML pages which form the web application interface are:

- **HomePage**: through this page, end users can choose what operation to carry out next, that is, send a transaction, send a query, or inspect the blockchain.
- **SendTransactionPage**: through this page, end users can easily insert the body of a transaction they wish to send and specify the sending mode for that transaction, that is, *sync*, *async*, or *commit*. For invocation transactions, they can choose their type as well, that is, *unicast*, *multicast*, or *broadcast*.
- **QueryPage**: through this page, end users can request a full list of all the existing smart contracts and the source code of a desired smart contract.
- **BlockchainPage**: through this page, end users can specify both a "minHeight" and a "maxHeight" in order to retrieve the desired set of contiguous block.

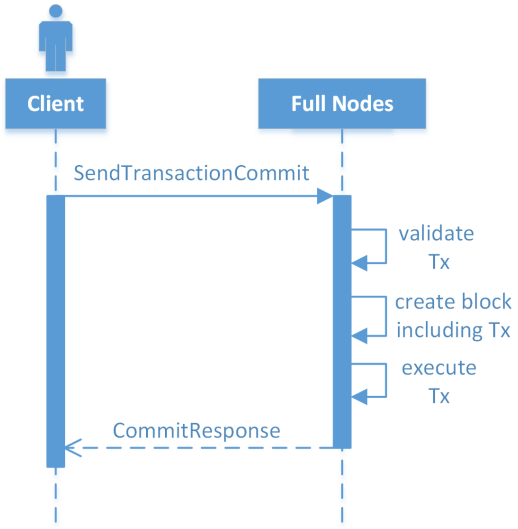
For a given transaction that needs to be sent, end users have to specify its body only. The transaction itself is properly built by clients which automatically insert all the other fields.



(a) The client sends a SendTransactionAsync request. The response is empty.



(b) The client sends a SendTransactionSync request. The response contains a boolean, stating whether the validation process has succeeded or not.



(c) The client sends a SendTransactionCommit request. The response contains the result of the transaction execution.

Figure 4.4: Sequence diagrams depicting the interaction between a client and the network of full nodes. The whole peer-to-peer network is depicted as a single entity (i.e. Full Nodes) for clarity purposes. Nonetheless, note the a client actually sends requests to a single full node. The interaction depends on the kind of HTTP request which is sent.

4.2.6 Certification authorities

Every CA in the hierarchy is provided with a certificate released by a CA from the upper level, except for the root, whose certificate is self-signed. Each one of them consists of a web server, which listens for two types of HTTP requests at port `3000`, namely, `GenerateCertificate` and `RetrieveCertificate`. End users (through clients) and full nodes can request a new certificate to leaves CAs through a `GenerateCertificate` request. Furthermore, they can also send a `RetrieveCertificate` request to a certain CA in order to require its certificate.

In general, the validation of a certificate X owned either by a end user or a full node is carried out as follows:

1. Thanks to the URI contained in X, a `RetrieveCertificate` request is made to the issuer of X in order to retrieve its certificate Y.

2. The public key contained in Y is used to verify the signature of X. If the signature is invalid or the certificate has expired, then X is invalid. Otherwise, another `RetrieveCertificate` request is made to the issuer of Y in order to retrieve its certificate.

3. The routine is repeated as long as all signatures are valid and all certificates have not expired, until a request is made to the root CA. If its certificate is valid and has not expired yet, then the certificate X is valid.

Finally, each CA keeps track of the certificates which it releases until they expire, in order to avoid the creation of two certificates for the same public key.

Chapter 5

Implementation

In this chapter we briefly discuss the system implementation [36]. In section 5.1 we provide an overview of the system components' implementation. In section 5.2 we describe how smart contracts can effectively perform delayed/periodic tasks and send transactions. Finally, in section 5.3 we discuss two technical features of the system which are implemented to enhance the overall security.

5.1 Implementation overview

In order to shorten the system implementation process, we decide to employ Tendermint as the core of each full node. Thus, all the aspects related to blockchain management and peer-to-peer connectivity are handled by Tendermint, which also expose a set of HTTP endpoints which can be used by clients and smart contracts to send transactions, query the application state, and inspect the blockchain through the HTTP requests described in subsection 4.1.3.

Furthermore, since this project is meant to be a proof-of-concept for proactive smart contracts, in this phase we substitute the hierarchy of certification authorities with a single CA, in order to further shorten the realization of the system.

5.1.1 Logic interpreter

The logic interpreter is written in Java. Regarding the communication with Tendermint Core, we rely on JTendermint, an ABCI server written in Java. Thus, the Server component described in subsection 4.2.4 is actually replaced by JTendermint. All the other components are implemented as *singletons*. When the logic interpreter is started, the following operations are performed:

1. All the components are statically initialized, except for the Controller and the Certificate Manager. Upon initialization, the Smart Contract Manager create two distinct sets, meant to store smart contracts and their *dummy* copies respectively, while the Key Manager generates the full node's key pair.

2. A JTendermint server is created, setting the Controller as a listener for messages sent by Tendermint Core. When launched, the JTendermint server waits for three connections to be opened by Tendermint Core, as described in subsection 2.3.2. The server runs on his own thread.

3. The Controller is initialized, which in turn initializes the Certificate Manager. The latter retrieves the full node's public key from the Key Manager and sends a `GenerateCertificate` request to the certification authority to obtain a proper certificate for the full node.

The logic interpreter is now ready to process both transactions and queries.

5.1.2 Certification authority & client

The certification authority is a simple web server written in Node[1]. When started, it waits for `GenerateCertificate` requests as described in subsection 4.2.6. It does not support `RetrieveCertificate` requests since there is not a hierarchy of CAs yet.

The client is a web application written in Node too. More precisely, the server side is actually written in Typescript[2], while the HTML pages described in subsection 4.2.5 are created and rendered through Pug[3]. When started, the client automatically generates a RSA key pair and sends a `GenerateCertificate` request to the certification authority to obtain a proper certificate.

5.2 Smart contracts

In order to be able to write smart contracts as logic programs within an application written in Java (i.e. the logic interpreter), we relied on multi-paradigm programming with *tuProlog*[4]. tuProlog is a lightweight implementation of the Prolog language which offers a so-called *Java API*, meant to allow developers to use a logic engine within a Java application.

[1] `https://nodejs.org/en/`
[2] `https://www.typescriptlang.org/`
[3] `https://pugjs.org/api/getting-started.html`
[4] `http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome`

A tuProlog logic engine establishes two predicate types, that is, *library* predicates, which *cannot* be modified or deleted at runtime, and *user-defined* predicates, which are modifiable at runtime through meta-programming. From a technical standpoint, each tuProlog engine is composed of a *theory*, meant to contain all the user-defined predicates, and a set of libraries, meant to contain all the library predicates. The theory makes up the dynamic KB, while the set of libraries makes up the static KB.

Each smart contract is thus a Java object which encapsulates a tuProlog logic engine, which is an object as well. The smart contract's source code is stored in the dynamic KB of the engine (i.e. it is stored in the engine as its theory). Each engine is provided with several libraries by default, which contain useful, "general-purpose" predicates. Moreover, each engine makes it possible to submit goals to the source code and retrieve the corresponding result, if any.

The Java API allows developers to *(i)* write new library predicates and add them to the static KB of a logic engine, *(ii)* remove libraries from the static KB of a logic engine. The body of new library predicates can be written directly in Java. Thus, we exploited this functionality to provide the logic engine of each smart contract with a set of library predicates, meant to enable smart contracts to send transactions and schedule both delayed and periodic task. In this context, the last statement actually means "schedule the delayed or periodic submission of a goal to the logic engine". Such predicates are as follows[5]:

- `self/1`
  `self(Identity)` allows to retrieve the identity of the smart contract.
  Template: `self(-Identity)`

- `owner/1`
  `owner(Identity)` allows to retrieve the identity of the smart contract's owner.
  Template: `owner(-Identity)`

- `sender/1`
  `sender(Identity)` is true if the smart contract is solving a goal contained in an invocation transaction. In this case, the predicate can be used to retrieve the identity of the transaction sender. Otherwise, this predicate is false.
  Template: `sender(-Identity)`

[5]The prefix "-" denotes an output term, while the prefix "+" denotes an input term.

- `now/1`
  `now(Time)` allows to retrieve the system global time. It is the one which makes smart contracts aware of time.
  Template: `now(-Time)`

- `send/2`
  `send(Identity_List, Goal)` allows smart contracts to send an invocation transaction with the specified `Goal` to one or more smart contracts, whose identities are specified in `Identity_List`. The `Identity_List` can be replaced by the anonymous variable "_" to send a transaction in broadcast mode.
  Template: `send(+Identity_List, +Goal)`

- `ack/2`
  `ack(Transaction, Result)` allows smart contracts to send an acknowledgement transaction for a certain invocation `Transaction` which has been issued to them by another smart contract, including the resolution `Result` in it.
  Template: `ack(+Transaction, +Result)`

- `delayed_task/2`
  `delayed_task(Delay, Goal)` submits the specified `Goal` to the logic engine after an initial `Delay`, which is expressed in seconds.
  Template: `delayed_task(+Delay, +Goal)`

- `when/2`
  `when(Global_Time, Goal)` submits the specified `Goal` to the logic engine when the system global time reaches the point in time `Global_Time`, expressed as Unix epoch. If the specified `Global_Time` is a point in the past, this predicate is false. This predicate is a variant of `delayed_task/2`.
  Template: `when(+Global_Time, +Goal)`

- `periodic_task/3`
  `periodic_task(Delay, Period, Goal)` submits the specified `Goal` to the logic engine periodically (i.e. once every `Period`) after an initial `Delay`. Both `Period` and `Delay` are expressed in seconds.
  Template: `periodic_task(+Delay, +Period, +Goal)`

The predicates which allow for the postponed or periodic submission of goals to the logic engine (i.e. `delayed_task/2`, `when/2`, and `periodic_task/3`) have been realized through Java's `ScheduledExecutorService`. Summing up,

all the smart contracts that are created by end users are equipped with these library predicates by default, which can be used but not modified or cancelled.

At last, a note on non-determinism. tuProlog engines supports the creation of random numbers through a subset of library predicates. In order to make the resolution process of goals strictly deterministic, we removed these library predicates from the static KB of each engine. As a result, the submission of some goal to a certain smart contract will *always* give the same result.

5.3 Further security enhancements

In this section, we describe two different techniques which decide to employ to deal with faulty logic interpreters and system attackers respectively.

5.3.1 Faulty interpreters tolerance

The peer-to-peer network of the system is composed by full nodes. Each one of them contains a copy of the Tendermint Core and a copy of our logic interpreter. Recall that the Tendermint consensus is BFT. Given n copies of the Tendermint Core participating in the consensus, the algorithm can tolerate up to $n/3 - 1$ of them exhibiting arbitrary behaviour. Unfortunately, this property does not hold for our logic interpreters. At the current state, a single malicious copy of the logic interpreter provided with a proper certificate could pretend to be a smart contract and forge an invocation transaction accordingly. This forged transaction would exceed the validation process carried out by other logic interpreters and would eventually be executed.

In such a scenario, the "victim" smart contract would be held accountable for the transaction execution and for any hypothetical damage caused to the system by it. Moreover, the next transaction sent by the victim smart contract would be invalid, as its nonce would not match up with the one expected by the Transaction Validator.

To overcome this problem, we exploit the "replicated" nature of smart contracts. Recall that even if end users perceive a certain smart contract as a single entity, there are actually n copies of it in the system, one in every full node, where n is the number of full nodes. This means that every time a smart contract sends a transaction, n copies of this transaction are actually sent. Each copy is endowed with the same `IDENTITY`, `BODY`, and `NONCE`, while having different `CERTIFICATE` and `SIGNATURE` from one another.

Thus, the solution to the problem is as follows. Each logic interpreter executes a certain transaction sent by a smart contract only if it receives at least a certain percentage x of its copies within a fixed amount of time, each

one of them endowed with a different `SIGNATURE`. With this approach, forged transaction are not executed even if sent multiple times, since each copy would have the same `SIGNATURE`. The default value of x is 67% and can be adjusted by the system owner as desired, before starting the logic interpreter.

This solution is made viable thanks to Tendermint, which makes each logic interpreter aware of the number of full nodes that are participating in the consensus algorithm, that is, the total number of transaction copies that will be sent for each authentic transaction generated by a smart contract.

5.3.2 Auditing

At the current state, invalid transactions are just rejected by full nodes, meaning that they are neither included in the blockchain nor executed. Since invalid transactions include the ones sent by hypothetical system attackers, it would be useful to store them in the blockchain in order to enable system administrators to consult them and take action against such attackers.

To this end, we introduce a new transaction type, namely, *audit* transactions, meant to keep track of invalid ones. These transactions are sent to Tendermint Cores by logic interpreters whenever a faulty transaction does not exceed the validation process. Each one of them contains the faulty transaction and the error which caused the failure of the validation process. More precisely, the body of an audit transaction is of type `audit(Transaction, Error)`. The possible errors are *no_identity_found* (i.e. the transaction is addressed to a smart contract that does not exist), *unexpected_nonce*, *invalid_transaction_signature*, *invalid_certificate_signature*, *certificate_expired*, *creation_timeout* (i.e. the operation carried out by the smart contract upon its creation caused a timeout), *invocation_timeout*, and *forged_transaction*. Audit transactions are addressed to a special smart contract created by the system at start-up, namely, *auditing smart contract*. This smart contract cannot be modified or destroyed. It is not provided with the library predicates described in section 5.2 since its only purpose is to store audit transactions in its dynamic KB, as a part of its source code. Thus, audit transactions can be perceived as invocation transactions issued to the auditing smart contract, which simply stores their bodies in its dynamic KB.

While following this approach, audit transactions are stored both in the blockchain and at application level thanks to the auditing smart contract. End users can retrieve the full list of audit transactions by sending a `Query` request to full nodes for the source code of the auditing smart contract.

Audit transactions suffer from the same problem of invocation transactions sent by smart contracts, that is, a faulty logic interpreter could forge an audit transaction a send it to the auditing smart contract. More precisely, a

faulty logic interpreter could *(i)* forge an invocation transaction with an invalid signature and *(ii)* package it into an audit transaction, specifying the error *invalid_signature*. The audit transaction would exceed the validation process and would eventually be executed. To overcome this problem, we employ the approach described in the previous section for audit transactions too. Thus, a certain audit transaction is executed by a logic interpreter only if it receives at least the fixed percentage x of the transaction's total copies, each one of them endowed with a different `SIGNATURE`.

Chapter 6

Validation

After the previous chapters, where we analysed and modelled the system to finally implement it, we now proceed with its validation. More precisely, we briefly discuss the requirement compliance of the system in section 6.1. Then, in section 6.2 we describe three use cases that can be put into practice in the context of BCTs thanks to our smart contracts.

6.1 Requirement compliance

The system adheres to all the requirements, both functional and non-functional. Moreover, it is even endowed with additional functionalities that were introduced during the design phase in section 4.1, such as *(i)* the support for invocation transactions in multicast and broadcast mode, *(ii)* the support for a form of "synchronous" communication between smart contracts, *(iii)* the adoption of RBAC to provide the system with a finer control over who is allowed to interact with a certain smart contract, and *(iv)* the possibility for end users to send transactions in three different modes, namely, *async*, *sync*, and *commit*.

6.2 Real-world use cases

In this section we describe three different use cases which can be realized within the context of BCTs by employing proactive smart contracts. All of these use cases require smart contracts to be reactive to time and able to perform asynchronous communication. Thus, they cannot be put into practice relying on "classic" smart contracts.

6.2.1 Periodic Payments

By adding support for a cryptocurrency as a distributed asset within the system, smart contracts would be able to carry out periodic payments thanks to their reactivity to time. In this case, the logic interpreter would keep track of the amounts of this cryptocurrency hold by each end user and each smart contract. Furthermore, the system would support an additional type of transaction, meant to send amount of cryptocurrency to either smart contracts or end users, namely, *transfer* transactions. Smart contracts would be endowed with an additional library predicate which would allow them to send these type of transactions, namely, `transfer(Recipient_Id, Amount)`. A *transfer* transaction addressed to a smart contracts would also cause the invocation of goal `receive(transfer(N))`, where `N` is the amount of cryptocurrency contained in the transaction. Note that adding support to the system for these new functionalities requires minimal effort.

Regarding the use case, let us suppose that some ACME company offers a service for which subscribed customers need to pay monthly. For this service, ACME accepts payments through our system, setting a service price of 10 units of cryptocurrency per month. Thus, in this scenario ACME would have its own smart contract, called *company contract* henceforth, and would use it to correctly (and automatically) manage monthly payments from customers. The *company contract*, whose source code is depicted in Listing 6.1, would keep track of the credit of all the subscribed customers and run a periodic task meant to withdraw 10 units of cryptocurrency from each customer credit once every 30 days. To achieve the latter functionality, the company would specify `periodic_task(0, 2592000, withdraw)`[1] as the initial goal to be submitted to the *company contract*. Furthermore, the *company contract* would allow customers to specify the identity of the smart contract which will send periodic payments on their behalf.

```
% Facts meant to keep track of which customer a
% certain smart contract pays for.
contract_to_customer(id_smart_contract_1, id_customerA).
contract_to_customer(id_smart_contract_2, id_customerB).

% Facts meant to store customers' credit, represented
% as units of cryptocurrency.
credit(id_costumerA, 7).
credit(id_costumerB, -2).

% Rule meant to allow customers to declare
```

[1] 2592000 seconds are equal to 30 days.

```
% the idenity of the smart contract which
% will make periodic payments on their behalf.
receive(bind_id(ContractId)) :-
   sender(Sender),
   credit(Sender, _), % check if the sender is a customer
   assert(contract_to_customer(ContractId, Sender).

% Rule used to accept payments from smart contracts
% and update the credit of the corresponding customer.
receive(transfer(Amount)) :-
   sender(Sender),
   contract_to_customer(Sender, UserId),
   retract(credit(UserId, Credit)),
   New_Credit is Credit + Amount,
   assert(credit(UserId, New_Credit)).

% Rule used to withdraw 10 units of cryptocurrency
% from all the customer balances.
withdraw :-
   findall(ID, credit(ID, _), IDs),
   withdraw(IDs).

withdraw([]).

% For the current ID, withdraw 10 units
% from the corresponding credit
withdraw([ID | Other_IDs]) :-
   retract(credit(ID, Amount)),
   New_Amount is Amount - 10,
   assert(credit(ID, New_Amount)),
   withdraw(Other_IDs).
```

Listing 6.1: The *company contract*'s source code.

Customers could create a smart contract to carry out automatic payments for the service they are subscribed to (source code in Listing 6.2). First, they would send an invocation transaction to the *company contract* meant to bind their identity to the one of their smart contract. Then, they would provide their smart contract with some amount of cryptocurrency and initialize periodic payments through an invocation transaction. More precisely, to carry out the latter operation they would retrieve the identity of the *company contract* (say, `sc7098`) and send an invocation transaction to their smart contract containing the goal `pay_periodically(sc7098, 2592000, 10)`.

```
% Rule to initialize a periodic payment to a certain recipient.
% First, a check is made in order to verify that the issuer of the
% invocation transaction is effectively the owner of the smart
% contract. Then, the "periodic payment" task is launched.
receive(pay_periodically(Recipient, Interval, Amount)) :-
   sender(Sender),
   owner(Sender),
   periodic_task(0, Interval, transfer(Recipient, Amount)).
```

Listing 6.2: The source code of a smart contract which allows its owner to carry out periodic payments.

With this approach, ACME would monitor costumers' credit and stop providing its service to customers as soon as their balance becomes negative. The realization of the functionalities offered by both of these smart contracts is possible thanks to the predicate `periodic_task/3`, which allows to schedule the periodic submission of a certain goal in a very simple way. As a result, both of these smart contracts and their functionalities *cannot* be realized in other BCTs, such as Ethereum.

6.2.2 Supply-chain management

Smart contracts capable of performing asynchronous interactions can be used to enforce the terms of a contract between two parties within the context of supply-chain management processes. We explain this statement with a real use case.

Let up suppose the manufacturer company ACME is the supplier of the company Globox for a certain product. For the shipment of this product, ACME draws up a contract with the shipping agency Cervice. The two parties negotiate for a price X, which ACME pays to Cervice in advance. Then, they establish some penalties for Cervice to be applied in case of delayed delivery. The contract states:

- The product must be delivered within 3 days
- If the delivery time is between 3 and 5 days, Cervice will refund 40% of the full amount X to ACME
- If the delivery time exceeds 5 days, Cervice will refund the full amount X to ACME

For security purposes, ACME and Cervice decide to rely on our system by creating a smart contract in charge of enforcing and fulfilling the aforementioned

rules. After the smart contract's creation, ACME sends a *transfer* transaction to it, in order to provide it with the necessary amount of cryptocurrency needed for the payment to Cervice. Note that both ACME and Cervice can be conceived as end users of the system, each one of them provided with a certificate and the related identity. The smart contract does not pay Cervice upfront in order to be capable of issuing refunds to ACME if necessary.

In order to make the smart contract aware of departure and arrival times of the product to be shipped, ACME and Globox act as follows. They both assign a smart device to one of their employees, which is able to read RFID tags and send transactions to the aforementioned smart contract. Smart devices are assumed to be endowed with a proper certificate in order to be able to interact with our system. Then, ACME attaches an RFID tag containing a UUID to the product.

Before assigning the product with UUID `‘‘ac3e’’`[2] to Cervice's courier, the ACME employee is assumed to read the RFID tag with his/her smart device, which automatically sends an invocation transaction to the smart contract. This transaction denotes the departure time of the product by carrying the term `departure(ac3e)`. When the smart contract receives this transaction, it schedules two delayed task meant to automatically trigger refunds when (and if) necessary. These task are delayed by 3 and 5 days respectively. The first one sends back 40% of the full amount X to ACME, while the second one sends back the remaining 60% of the full amount. These refunds effectively occur only if the product has not reached destination yet.

When the product reaches its destination, the Globox employee reads the RFID tag with his/her smart device, which automatically sends an invocation transaction to the smart contract as well. This transaction denotes the arrival time of the product by carrying the term `arrival(ac3e)`. When receiving this transaction, the smart contract pays Cervice the remaining amount of X, if any. Listing 6.3 contains the smart contract's source code.

The realization of this use case is made possible thanks to the predicate `delayed_task/2`, which enables the smart contract to automatically send back refunds to ACME when necessary, without waiting for the product to be effectively delivered. Thus, as the previous one, this use case *cannot* be realized in other BCTs, such as Ethereum.

[2]For the sake of clarity, in this example, we use a 4-character string instead of a proper UUID, since UUID are quite verbose

```
% Facts which store the necessary identities.
acme_id(o38bn27y4).
cervice_id(d92jf4dfs).
acme_smart_device_id(g21vfjd4c).
globox_smart_device_id(y5dv7pc21).
shipment_cost(50).

receive(departure(UUID)) :-
   sender(Sender),
   acme_smart_device_id(Sender),
   shipment_cost(Amount),
   assert(to_pay(UUID, Amount)), % remember to pay
   delayed_task(3 * 86400, partial_refund(UUID)), % 3-day delay
   delayed_task(5 * 86400, total_refund(UUID)). % 5-day delay

receive(arrival(UUID)) :-
   sender(Sender),
   globox_smart_device_id(Sender),
   retract(to_pay(UUID, Amount)), % if this rule is still present...
   cervice_id(Id), % ...then remove it...
   transfer(Id, Amount). % ...and pay Cervice

partial_refund(UUID) :-
   retract(to_pay(UUID, Amount)), % if this rule is still present...
   Refund is Amount * 0.40,
   acme_id(Id),
   transfer(Id, Refund), % ...refund 40% to ACME...
   Remaining is Amount * 0.60,
   assert(to_pay(UUID, Remaining)). % ...and store the remaining

total_refund(UUID) :-
   retract(to_pay(UUID, Remaining)), % if this rule...
   acme_id(Id), % ...is still present, remove it and...
   transfer(Id, Remaining). % ...refund the full amount to ACME
```

Listing 6.3: The source code of the smart contract created by ACME. Upon a product departure, it schedules two delayed task meant to perform automatic refunds if necessary. Upon a product arrival, it pays Cervice through a *transfer* transaction only if the delivery time does not exceed 5 days

6.2.3 Automatic auctions

Proactive smart contracts can also be employed for the realization of *contract net-based* applications within the context of a BCT. The term *contract net* derives from the homonymous protocol [3] which is generally employed within the scope of Multi-Agent Systems in order to let an agent propose some task to one or more other agents, possibly competing to be the best one for its fulfilment. In such situation, those agents would then form a so-called "contract net". When an agent needs help for the fulfilment of a task, it announces it to the contract net along with a point in time T, playing the *manager* role. The latter represent the time threshold beyond which the *manager* will no longer accept bids. When receiving this announce, the other agents can decide whether to make a bid for it, playing the role of (potential) *contractors*. The manager collects bids until T and then chooses the best bid, assigning the task to the corresponding contractor. In general, agents can announce multiple tasks and choose multiple bids for their fulfilment.

Regarding our system, an interesting use case related to contract nets is represented by automatic auctions. Let us suppose that the system makes it possible to keep track of the ownership of some *products* through the underlying blockchain. Both end users and smart contracts can own products and are able to claim ownership when required. They are also able to transfer ownership of their products to other end users or smart contracts by means of special transactions, namely, *transfer ownership* transactions. Note that adding support to the system for this new functionality requires minimal effort.

A *seller* smart contract could set up an auction for a predefined amount of time in order to sell its products. First, it would open the auction and schedule a delayed task meant to close it when desired. Then, it would announce the auction opening through an invocation transaction in broadcast mode, specifying the products for sale and the auction deadline. When receiving the announce, each *buyer* smart contract would establish whether it is interested in one or more of the products and send back proposals for them; each proposal would contain the product of interest and the amount of cryptocurrency that the smart contract is willing to pay for it. The *seller* smart contract would gather proposals as long as the auction is still running. Upon auction end, it would choose the best proposal for each one of its products and alert the corresponding issuers (called *winning buyers* henceforth) with an invocation transaction. Each *winning buyer* would then send a transfer transaction to the *seller* containing the proposed amount of cryptocurrency. When receiving a payment from a *winning buyer*, the *seller* smart contract would issue a *transfer ownership* transactions to it, finalizing the deal.

Example of source codes of both *sellers* and *buyers* can be found in List-

ing 6.4 and Listing 6.5 respectively. As in the previous example, the predicate `delayed_task/2` is essential for the realization of this use case, since it makes it possible for the auction to be automatic. Thus, this use case *cannot* be realized in other BCTs.

```
auction_duration(86400).

% Products for sale.
selling([product1, product2, product3]).

receive(open_auction) :-
   sender(S),
   owner(S),
   retractall(proposal(_, _, _)), % remove previous proposals...
   retractall(winner(_)), % ...and winners
   assert(auction_is_open), % the auction is now open
   auction_duration(Duration), % schedule a task meant to...
   delayed_task(Duration, close_auction), % ...close the auction
   now(T),
   selling(Products),
   send(_, ongoing_auction(deadline(T + Duration), Products)). %
      broadcast

receive(proposal(Amount, Product)) :-
   auction_is_open, % if the auction is still open
   selling(Products),
   member(Product, Products),
   sender(Sender),
   assert(proposal(Sender, Amount, Product)). % store the proposal

receive(transfer(Amount)) :-
   sender(Sender),
   winner(Sender), % if a winner pays...
   proposal(Sender, Amount, Product), % ...for its product...
   transfer_ownership(Product, Sender), % ...transfer ownership...
   retract(selling(Products)), % ...and update the product list
   delete(Product, Products, Updated_List),
   assert(selling(Updated_List).

close_auction :-
   retract(auction_is_open), % the auction is now closed
   selling(Products),
   sell(Products). % check proposals and sell products
```

```
sell([]).

% Find the best proposal for Product and invoke its issuer.
sell([Product | Other_Products]) :-
   findall(Amount, proposal(_, Amount, Product), Offers),
   max(Offers, Max_Amount),
   proposal(Sender, Product, Max_Amount), % the best proposal
   assert(winner(Sender)), % remember the "winner"
   send(Sender, best_proposal(proposal(Product, Max_Amount))),
   sell(Other_Products).

% If no proposals were made for Product, check the next product.
sell([Product | Other_Products]) :-
   findall(Amount, proposal(_, Product, Amount), Offers),
   length(Offers, L),
   L = 0,
   sell(Other_Products).
```

Listing 6.4: The *seller*'s source code. This smart contract is able to open an auction for a list of products, broadcasting an annouce for it. It gathers proposals from *buyers* until the auction is open. Upon auction end, it chooses the best proposal for each of its products and alerts the corresponding *buyers*. When the payment with respect to a certain product is received, it finalizes the deal with a *transfer ownership* transaction, sending it through the library predicate `transfer_ownership/2`.

```
% The products of interest for this smart contract.
interest(product1, 20).
interest(product2, 35).

receive(ongoing_auction(deadline(Deadline), Products)) :-
   now(T),
   T < Deadline, % If the auction is still running
   evaluate_products(Products). % evaluate the products

receive(best_proposal(proposal(Product, Amount)) :-
   sender(Sender), % If my proposal is the best
   retract(proposal_made(S, Product, Amount)),
   transfer(Sender, Amount). % pay the amount

evaluate_products([]).

evaluate_products([Product | Other_Products]) :-
   not(interest(Product, _)), % If not interested
   evaluate_products(Other_Products). % evaluate the next product
```

```
evaluate_products([Product | Other_Products]) :-
   interest(Product, Amount), % If interested
   sender(Sender),
   send(Sender, proposal(Product, Amount)), % make a proposal
   assert(proposal_made(Sender, Product, Amount)).
```

Listing 6.5: The *buyer*'s source code. This smart contract stores the products of its interest along with the amount of cryptocurrency which it is willing to pay for each one of them. Whenever a seller sets up an auction for a list of products and annouces it through a broadcast, the buyer checks the list and sends a proposal back to the seller for each product of interest. If one of its proposals is chosen by the seller (i.e. the buyer is invoked with the goal `best_proposal(P)`), the buyer transfers to the seller the amount of cryptocurrency proposed.

Chapter 7

On smart contracts' autonomy

In this chapter we discuss smart contracts' autonomy, concerning both our proposed "logic contract" notion and future declinations of the smart contract notion, which may possibly be influenced by other computational or programming models. In section 7.1 we recap the work that has been carried out to implement these logic, proactive smart contracts, pointing out our final goal. In section 7.2 we provide a comparison for them with *actors* and *agents*, highlighting similarities and differences.

7.1 Proactive smart contracts

At the beginning of this work, we analysed smart contracts and their modern implementation, taking Ethereum as our main reference. We studied the limitations deriving by their OOP-like nature, stating how the choice of modelling them as passive, synchronous objects may limit the amount of scenarios where smart contracts would be useful or of interest. Stemming from such remarks, we rethought smart contracts as proactive entities with one clear motivation in mind, that is, making them actually *autonomous* entities, capable of *acting* in order to reach or maintain the goals and (contractual) terms they have been created for. To reach this goal, we endowed smart contracts with the ability of interacting *asynchronously* with each others in the first place. By combining this feature with *time-awareness* we made smart contracts reactive to time, and thus able to perform postponed or periodic computations over time. Finally, to move a first step towards autonomy, we gave smart contracts the ability of performing an initial operation when created. Thanks to this capability, smart contracts can "initialize" themselves autonomously, carrying out certain operations according to their own internal logic.

In the realization process, we came across several problems concerning the system security, which were solved by adopting a private blockchain and rely-

ing on certification authorities. Furthermore, we employed RBAC and other security enhancements, described in section 5.3, which made the system resistant to faulty nodes at a certain extent. At the current state, we could not find any further security flaw in the system.

The real word use cases described in section 6.2 showed how our smart contracts can effectively enforce the terms of their physical counterparts and strictly regulate the interaction between their parties. They mediate this interaction in an "active" way, ensuring the fulfilment of terms and carrying out payments on behalf of physical users. Furthermore, their logic nature enhances the observability of their source code, making their business logic easier to understand, update, and verify.

Our current implementation paves the way towards the achievement of our final end, which consists in the realization of fully autonomous smart contracts, able to observe the behaviour of their parties, detect if some violation is committed with respect to the contractual terms, and act against such violations. To achieve this end, the logic nature of smart contracts could be exploited to make them capable of planning and reasoning, in order to *(i)* proactively and contextually infer the best way to enforce the real world condition/agreements they should guarantee, *(ii)* plan a course of action, and finally *(iii)* act, executing such plan.

7.2 Comparison with actors and agents

In the field of computer science, system components are usually modelled (and implemented) as either *objects*, *actors*, or *agents*, which are different classes of computational entities.

Objects are passive entities. They encapsulate a state, made up by a set of variables (called *fields*), and expose a set of *methods* that can be invoked through *method-calling*. These methods are meant to modify their state in a controlled way. Since objects are passive, they do *not* encapsulate a thread of control, meaning that their methods are only invoked by external entities.

Actors are active/reactive entities. They encapsulate a state, just like objects, and can be invoked through *message-passing*. Their methods can be conceived as *message handlers*, each one of them consisting in a set of operations to be carried out when a certain message is received. They are *active* in the sense that they do encapsulate the thread of control. At the same time, they are *reactive* in the sense that they perform some kind of computation only when they receive a message. Thus, actors are actually *reactive to messages*.

Agents are *autonomous* entities. By definition, they *(i)* encapsulate the

thread of control, *(ii)* have total control over their own internal state, *(iii)* *cannot* be invoked by external entities, and *(iv)* encapsulate a *criterion* to self-govern their control flow. These properties make up the essential feature of agents, that is, *autonomy*. From this feature, many others features stem. Agents are *proactive*, in the sense that they can use their *criterion* to make something happen instead of necessarily wait for something to happen. Agents are *situated*, that is, they are strictly coupled with the context where they live and interact. Thus, their actions are modelled depending on the representation of context where they take place. Thanks to their *situatedness*, agents are *reactive to changes* to the context. Furthermore, agents are *interacting*, meaning that they can interact with other agents by means of *communication actions*. Finally, agents are *goal/task oriented* entities, meaning that they build possible plans of actions to either reach a certain state of the world (i.e. a *goal*) or bring an activity (i.e. a *task*) to an end.

These features make up the notion of *weak agent*, which is opposed to the notion of *strong agent* [7]. Strong agents are conceptually endowed with mental components such as *beliefs*, *desires*, *intentions*, *goals*, and *plans*. Strong agents may be *intelligent*, and use their intelligence as the *criterion* for self-governance.

Our smart contracts are placed in between the notion of actor and the notion of agent. They are more than actors since *(i)* they are proactive and *(ii)* they are reactive to both messages sent to them, i.e. invocation transactions, and *time*. More precisely, the notion of actor does not necessarily include *time awareness*, making (default) actors incapable of performing postponed or periodic computation. Nonetheless, even if our smart contracts do exhibit a first form of proactivity through the operation carried out upon their creation, they cannot be considered as agents since they (still) lack their fundamental feature, that is, *autonomy*. They do not have total control over their internal state, since they cannot refuse invocation transactions as agents can refuse *communication actions*. Furthermore, even if they can be conceived as situated entities where the context is represented by the underlying blockchain, they are not reactive to changes to the context in general. As a result our smart contracts are incapable of planning and reasoning about their context to automatically achieve goals and fulfil tasks submitted to them.

Chapter 8

Conclusions

In this last chapter, we firstly summarize the work that has been done with this master thesis in section 8.1. Finally, we discuss some possible future works that could be carried out to enhance the system implementation in section 8.2.

8.1 Summary

In chapter 2, we provided a technical background for what concerns blockchains, smart contracts, and logic programming. We also described Tendermint in detail, being the consensus engine we relied on for the realization of our system. In chapter 3, we described our vision of *proactive* smart contracts, defining a set of requirements for the system we wished to realize. Stemming from such requirements, we carried out a feasibility study which led us to choose the adoption of a private blockchain in advance. In chapter 4, we modelled the system and its components, describing their structure, interaction, and behaviour. We also chose to endow the system with additional features meant to further enhance it. In chapter 5, we provided an overview on the system implementation, describing the main elements of interest. In chapter 6, we validated the system, describing three different use cases in which our smart contracts can be used to actively execute the terms of a contract. Finally, we discussed our smart contracts' autonomy in chapter 7, and we compared their implementation with the notion of *actor* and *agent*.

This top-down methodology allowed us to *(i)* both detect and analyse the problems arising from our idea, *(ii)* carefully model the system and all its functionalities, and *(iii)* implement the system with ease.

Summing up, the system offers a first support for the creation and execution of proactive logic smart contracts on top of a private blockchain. As already stated in section 6.1, it has been developed to fulfil all the requirements grouped in section 3.2. Moreover, it is endowed with additional features that we decided

to implement *(i)* as a result of our design choices described in subsection 4.1.1 and *(ii)* to enhance the overall system security, as described in section 5.3. The CAs hierarchy proposed in subsection 4.1.2 has not been implemented. Nonetheless, its actual implementation would require minimal effort.

Ultimately, we are overall satisfied with the work that has been carried out with this thesis.

8.2 Future works

The system has been implemented as a proof-of-concept for our idea of proactive smart contracts. For this reason, there are several works that could be carried out to *(i)* add new functionalities to the system and *(ii)* modify some existing functionalities in order to enhance the system under different points of view, from security and modularity to user experience. These works are as follows:

- **Smart contracts**: one of the main future works concerns smart contracts. Their current implementation could be enhanced in order to shorten the gap with the notion of *agent*. More precisely, they could be endowed with the ability of *(i) refusing* invocation transactions and *(ii)* being reactive to changes to the context (i.e. changes to the blockchain). With the latter ability, they would be effectively capable of automatically planning and reasoning about their "world".

- **Cryptocurrency**: the introduction of a cryptocurrency would make the system a valid candidate for the realization of different applications, as the ones described in section 6.2.

- **Better control over certificates**: at the current state, the certification authority releases a new certificate every time an entity, either a end user or a full node, requests one through a `GenerateCertificate` request. No controls are made on such a request and there is no upper bound to the number of certificates which can be released. In a real deployment of the system, an upper bound n would be presumably set in order to gain control over the number of system entities. If so, a malicious end user could create a *denial of service* attack by requesting n certificates for n different public keys, preventing other entities from obtaining a valid certificate and from interacting with the system. Note that malicious end users would be able to perform this attack since no controls are made on `GenerateCertificate` requests.

This problem could be faced introducing a registration phase, which both end users and full nodes must perform in order to obtain a proper certificate. This phase would allow the certification authority to uniquely identifying each entity, preventing the assignment of two or more certificates to the same entity.

- **CAs hierarchy**: at the current state, the system is provided with a single certification authority, which is in charge of assigning certificates to all entities, both end users and full nodes. One possible future work consists in the actual implementation of the CAs hierarchy described in subsection 4.1.2. The presence of a hierarchy would mitigate the *single point-of-trust* problem and make the system employable as a trusted platform shared by multiple, mutually-untrusted organizations. More precisely, given the presence of a root-CA, each organization could set up its own sub-CA within the system, meant to assign certificates to their members only. The system would act as a trusted intermediary among the organizations, where their members could create and manage smart contracts meant to regulate inter-organization interaction.

- **Clients enhancement**: at the current state, clients do not store the end user's key pair and certificate persistently. This means that every time a client is started, a new key pair is generated *and* a new certificate is requested to the certification authority. In a real system deployment where the total number of certificates that can be released is limited, the current client implementation would not be viable. Thus, one possible future work consists in enhancing the management of both keys and certificates so to store them persistently over times.

  Furthermore, when end users send a `RetrieveBlocks` request to a full node, the client shows the blocks content in JSON format. Thus, clients could be enhanced even from the user experience point of view, displaying the blocks content in a more understandable way.

- **IP manager**: at the current state, the IP addresses of both the full node and the certification authority are hard-coded within the client. As a result, a client always interacts with the same full node and requests certificates to the same certification authority. Given a real deployment where a client interacts with the full node X, if X crashes the client would no longer be able to send transactions to the system (i.e. to other full nodes). To solve this problem, a new component in charge of tracking full nodes' IP could be added to the system. This component would be an *IP manager*. Full nodes would register their IP to the IP manager and clients would retrieve the full list of IPs from it periodically. Thus,

clients would use these IPs in a *round-robin* fashion in order to interact with the system and balance the amount of transactions sent to each full node. If one full node crashes, another IP can be used. Furthermore, given the presence of a CAs hierarchy, the IP manager could be used to keep track of CAs' IPs too, enabling clients and full nodes to retrieve them when necessary.

- **Logic interpreter splitting**: at the current state, the logic interpreter of each full node is in charge of both *(i)* validating transactions, through the Transaction Validator, and *(ii)* executing them, through the Smart Contract Manager. To enhance system modularity and adhere to the *single responsibility principle*, each logic interpreter could be split in two components, in charge of validating and executing transactions respectively.

Bibliography

[1] John Alan Robinson. "A machine-oriented logic based on the resolution principle". In: *Journal of the ACM (JACM)* 12.1 (1965), pp. 23–41.

[2] Robert Kowalski. "Predicate logic as programming language". In: *IFIP congress*. Vol. 74. 1974, pp. 569–544.

[3] Reid G Smith. "The contract net protocol: High-level communication and control in a distributed problem solver". In: *IEEE Transactions on computers* 12 (1980), pp. 1104–1113.

[4] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the Presence of Partial Synchrony". In: *J. ACM* 35.2 (Apr. 1988), pp. 288–323. ISSN: 0004-5411. DOI: 10.1145/42282.42283. URL: http://doi.acm.org/10.1145/42282.42283.

[5] Nick Szabo. "Smart contracts". In: *Unpublished manuscript* (1994).

[6] David Ferraiolo, Janet Cugini, and D Richard Kuhn. "Role-based access control (RBAC): Features and motivations". In: *Proceedings of 11th annual computer security application conference*. 1995, pp. 241–48.

[7] Michael Wooldridge and Nicholas R Jennings. "Intelligent agents: Theory and practice". In: *The knowledge engineering review* 10.2 (1995), pp. 115–152.

[8] Leslie Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[9] Krzysztof R Apt. "The logic programming paradigm and prolog". In: *arXiv preprint cs/0107013* (2001).

[10] Adam Back. *Hashcash - A Denial of Service Counter-Measure*. http://www.hashcash.org/papers/hashcash.pdf. 2002.

[11] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Trans. Comput. Syst.* 20.4 (Nov. 2002), pp. 398–461. ISSN: 0734-2071. DOI: 10.1145/571637.571640. URL: http://doi.acm.org/10.1145/571637.571640.

[12] John R Douceur. "The sybil attack". In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 251–260.

[13] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. `https://bitcoin.org/bitcoin.pdf`. 2008.

[14] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, eds. *Replication: Theory and Practice*. Berlin, Heidelberg: Springer-Verlag, 2010.

[15] Rachid Guerraoui et al. "The Next 700 BFT Protocols". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: ACM, 2010, pp. 363–376. ISBN: 978-1-60558-577-2. DOI: `10.1145/1755913.1755950`. URL: `http://doi.acm.org/10.1145/1755913.1755950`.

[16] Joao Sousa and Alysson Bessani. "From Byzantine consensus to BFT state machine replication: A latency-optimal transformation". In: *Dependable Computing Conference (EDCC), 2012 Ninth European*. IEEE. 2012, pp. 37–48.

[17] Vitalik Buterin et al. "A next-generation smart contract and decentralized application platform". In: *white paper* (2014).

[18] Jae Kwon. "Tendermint: Consensus without mining". In: *Draft v. 0.6, fall* (2014).

[19] Diego Ongaro and John K Ousterhout. "In search of an understandable consensus algorithm." In: *USENIX Annual Technical Conference*. 2014, pp. 305–319.

[20] *Tendermint Images*. `https://tendermint.com/docs/introduction/introduction.html`. 2014.

[21] Gavin Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151 (2014), pp. 1–32.

[22] Melanie Swan. *Blockchain: Blueprint for a new economy.* " O'Reilly Media, Inc.", 2015.

[23] Marko Vukolić. "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication". In: *International Workshop on Open Problems in Network Security*. Springer. 2015, pp. 112–125.

[24] Richard Gendal Brown et al. "Corda: An introduction". In: *R3 CEV, August* (2016).

[25] Andrew Miller et al. "The Honey Badger of BFT Protocols". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 31–42. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978399. URL: http://doi.acm.org/10.1145/2976749.2978399.

[26] Arvind Narayanan et al. *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.

[27] Ryan Osgood. "The Future of Democracy: Blockchain Voting"'. In: *COMP116: Information Security* (2016).

[28] Serguei Popov. "The tangle". In: *cit. on* (2016), p. 131.

[29] Josh Stark. *Making Sense of Blockchain Smart Contracts*. Ed. by coindesk.com. https://www.coindesk.com/making-sense-smart-contracts/. June 2016.

[30] Christian Cachin and Marko Vukolić. "Blockchains consensus protocols in the wild". In: *arXiv preprint arXiv:1707.01873* (2017).

[31] Lin Chen et al. "On security analysis of proof-of-elapsed-time (poet)". In: *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer. 2017, pp. 282–297.

[32] *Proof of stake*. https://en.bitcoin.it/wiki/Proof_of_Stake. 2017.

[33] Zibin Zheng et al. "An overview of blockchain technology: Architecture, consensus, and future trends". In: *Big Data (BigData Congress), 2017 IEEE International Congress on*. IEEE. 2017, pp. 557–564.

[34] Elli Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. ACM. 2018, p. 30.

[35] Giovanni Ciatto et al. "From the Blockchain to Logic Programming and Back: Research Perspectives". In: *WOA 2018 – 19th Workshop "From Objects to Agents"*. Ed. by Massimo Cossentino, Luca Sabatucci, and Valeria Seidita. Vol. 2215. CEUR Workshop Proceedings. Sun SITE Central Europe, RWTH Aachen University, June 2018, pp. 69–74.

[36] Additional material: https://github.com/maffone/proactive-logic-smart-contracts.