UCLouvain

École polytechnique de Louvain

Use of the blockchain to ensure cloud data integrity in the frame of renewable energy communities

Authors: **Sacha Colette, Louis Colin**
Supervisor: **Axel Legay**
Readers: **Eduard Baranov, Emmanuel De Jaeger, Thibaut Piraux, Etienne Riviere**
Academic year 2020–2021
Master [120] in Electrical Engineering

Acknowlegment

Foremost, we would like to express our sincere gratitude to our supervisor, Pr Axel Legay for his patience and knowledge. His guidance and feedback helped us during this all year of work.

Besides our advisor, we would like to thanks the WeSmart company for there trust and enthusiasm. We would especially express our gratitude to Francois Bordes and Thibaut Piraut who took the time to answer our questions concerning the operation and the needs of the company.

Contents

Part I

Introduction

WeSmart is an enterprise that handles a cloud platform collecting data from smart meters and connected devices since 2018. The company aggregates different types of data: electricity consumption, air quality, temperature, humidity, pressure, water temperature, etc... It is also developing analytic algorithms in order to compare and interpret data. The result of the analysis is displayed online and on mobile devices for customers to monitor, understand and manage their energy and other critical environmental parameters [45].
WeSmart is also closely related to the sector of energy communities. In 2020, the company launched the energy community called "Greenbizz.energy". For this project, 943 solar pannels have been placed on the roof of the Greenbizz company[1]. Their role is to power companies but also private houses [92] [62].

In the frame of this work, WeSmart asked us to improve their current implementation thanks to the use of a blockchain. The objective was double : to guarantee the integrity of the data in the cloud and to explore the possibilities of performing energy trading between the participants. During this year, we first had to correspond with the WeSmart company in order to understand their current implementation and define their needs as much precisely as possible. This task being done, we performed a state of the art of the research domain. With those tools in hand, we then found and submitted a practical solution to the company. The goal was to validate the global solution but also the tools that we thought to be the most appropriate for the solution. After the validation of the company, we implemented this solution and finally submitted it to WeSmart.

The first part of this work presents concepts and state of the art solutions to answer these two objectives. The blockchain and microgrid concepts are introduced in the chapters 1 and 2 while chapter 3 and 4 respectively present the state of the art of the blockchain use for energy trading and to ensure cloud data integrity. A conclusion of this first part is that due to legal constraints it is nowadays not

[1]Greenbizz, located in Brussels, provides businesses with surfaces and services to create and develop their green, sustainable or environmentally related projects

possible to directly use the blockchain for energy trading. Our solution focuses then on data integrity checking.

More precisely, WeSmart currently stores their consumer related data in a cloud. However, there is no way to verify that those data are not corrupted when they are retrieved from the cloud. The proposed solution is an additional security layer that runs independently of the existing solution and whose goal is to corroborate the integrity of the data stored in the cloud. This solution is explained in the second part of this work. Chapter 5 explains the principle of the solution, the integration in the WeSmart architecture and the different tools available for the implementation. Chapter 6 clarifies the Hyperledger Sawtooth tool used for the implementation of the blockchain. Chapter 7 presents the results and performances of the solution. Finally, chapter 8 concludes by pointing the limitations of our solution and proposes improvement ideas.

Chapter 1

Blockchain concept

This chapter presents a review of the blockchain concept which will be used in all the remaining parts of this work. Note that an explanation of the digital signature concept, cryptographic hash function and the different kinds of failures that can happen in a distributed system is performed in the chapter 9 of the annexes. Those concepts are important in order to understand this report.

[20] The concept of blockchain started from an article of Satoshi Nakamoto in 2008 [4]. This concept came to fame as part of a proposal for Bitcoin (₿ or BTC), with the aim to create peer-to-peer money without centralized banks. The goal was multiple: avoid the need of trust in a third party, reduce transaction costs (bureaucracy) and anonymize transactions. However, the Bitcoin white paper did not come out of thin air, and P2P networks were not a new phenomenon. They are rooted in the early history of the computer and internet, building on decades of research of computer networks, cryptography, and game theory [33].
The major breakthrough of Bitcoin was to introduce a novel solution to the age-old human problem of trust. The underlying blockchain technology allows to trust the outputs of the system without trusting any actor within it. People and institutions who do not know or trust each other, reside in different countries, are subject to different jurisdictions, and have no legally binding agreements with each other, can now interact over the Internet without the need for trusted third parties like banks, internet platforms, or other types of clearing institutions. To do so, Bitcoin implemented a distributed network where each participant within the network maintains, approves, and updates new entries. The system is controlled not only by separated individuals, but by everyone within the network. Each member ensures that all records and procedures are in order, which results in data validity and security. Parties that do not necessarily trust each other are so able to reach a common consensus.
In the blockchain architecture, all computers in the network hold an identical copy

of the ledger of transactions which acts as a single point of reference. This ledger, also called blockchain, maintains a continuously growing list of transaction data records, contained in blocks. Each block includes the cryptographic hash of the prior block in the blockchain, linking one block with another into a chain of blocks, which guarantees the integrity of the previous block all the way back to the first one (the genesis block). In order to change the content of that ledger and create a new block, network users need to reach a mutual agreement, also referred to as consensus. The blockchain can, therefore, be described as a shared, trusted, public ledger of transactions, that everyone can inspect, but that no single user controls.

This architecture can be made in opposition with the traditional centralized architecture of the World Wide Web which uses a client-server network. In this case, the server keeps all the required information in one place. This method facilitates the update because the server is a centralized database controlled by a number of administrators with permissions. However, this client-server architecture is more prone to hacks and data leaks since the data are stored in one place. Moreover, this kind of architecture requires the trust in a third party [50]. With data being stored in several locations at once, information becomes more difficult to tamper with, whilst being available everywhere.

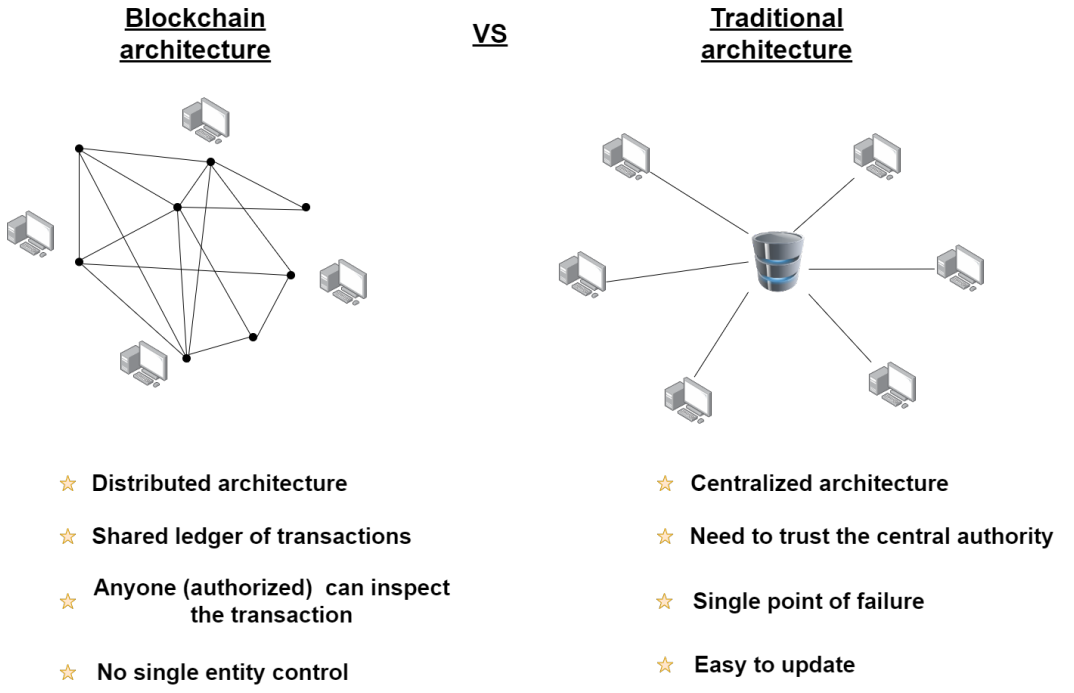


Figure 1.0.1: Comparison of the blockchain architecture and the traditional architecture of the World Wide Web

This new form of distributed data storage and management also avoids the double-spending problem (explained in the section "conflict and security" of the section 1.6.1) of existing value transfer over the internet. Ideas around cryptographically secured P2P networks have been discussed in the academic environment in different evolutionary stages, mostly in theoretical papers, since the 1980s. However, the Bitcoin was the first practical implementation of a P2P network that managed to avoid the double-spending problem.

1.1 Evolution of blockchain [34]

The next parts introduce the blockchain concept and begin with the evolution of the blockchain.
Three generations of blockchain technology are often considered in the blockchain evolution. Those are defined by Melanie Swan as [11] : blockchain 1.0 (programmable currency), blockchain 2.0 (programmable finance) and blockchain 3.0 (programmable society).

- Blockchain 1.0 (programmable currency): The blockchain technology appeared with the birth of Bitcoin. At this moment, its initial application scope is completely concentrated on digital currencies. The advent of programmable currencies has made possible the direct flow of value across the internet which brought a brand-new digital payment system. In this system, users can perform barrier-free digital currency transactions or cross-border payments. Moreover, the blockchain has the characteristics of decentralization, immutability, and trust, which can guarantee the security and reliability of transactions. The emergence of this digital currency has strongly impacted the traditional financial system.

- Blockchain 2.0 (programmable finance): Affected by digital currency, the application scope of blockchain technology has expanded to other financial fields. While programmable currency realises the decentralization of currency transactions, programmable finance can achieve the decentralization of the entire financial market, which is the next important link for the development of blockchain technology. Unlike using the blockchain as a support platform for virtual currencies, the core concept of blockchain 2.0 is to use the blockchain as a programmable distributed credit infrastructure to support the application of smart contracts (see section 1.8.1). Thanks to them, the application range of the blockchain has expanded from a single currency field to other financial fields involving contract functions. The contents of these transactions include real estate contracts, intellectual property rights, and debt certificates.

- Blockchain 3.0 (programmable society): With the further development of blockchain technology, the characteristics of decentralization and trust have made the application of blockchain gradually exceed the financial field. Blockchain 3.0 not only extends applications to social governance areas such as identity authentication, auditing, arbitration, bidding, but also includes industries, culture, sciences and arts. By solving the problem of distrust, blockchain technology provides a universal solution, that is, no longer establish credit and share information resources through third parties, thereby improving the operating efficiency and overall level of the entire field.

1.2 Public, private and consortium blockchains [61], [65], [81]

A blockchain can be of three types: public, private or consortium. Note that the public and private blockchains are the most widespread but a quick explanation of the consortium blockchain is performed for the sake of precision. Here is an explanation of the three:
A public blockchain is a permissionless blockchain. Anyone can join the blockchain network, meaning that they can read and write the distributed ledger. They can also take part in the consensus process.
On the other hand, a private blockchain is a permissioned blockchain. Private blockchains work based on access controls which restrict the people who can participate in the network.
Halfway between private and public blockchain, there is the consortium blockchain. Instead of having a single entity which decides who can join the blockchain, there are a few nodes that take this decision[39].

Here are the similarities that those blockchains have in common:

- They are decentralized and distributed which means that each node in these blockchains has a complete replica of the ledger. Because of the restricted number of nodes of the private blockchain, it can be qualified as partially decentralized compared to the fully decentralized public blockchain.

- All work as an append-only ledger where the records can be added but can not be altered or deleted. Hence, these are called immutable records. However, because the immutability of the ledger is based on the decentralization, a public blockchain is considered as more secure than the private blockchain. Indeed, due to the higher number of nodes in the public blockchain, it is

nearly impossible for 'bad actors' to attack the system and gain control over the consensus network.

Here are their differences:

- Private blockchains have less participants in their networks than public blockchains.
- In a public blockchain, anyone can take part by verifying and adding data to the blockchain. In private blockchains, only authorized entities can participate and control the network.
- Consensus algorithms are different in public and private blockchains.
- A public blockchain can not compete with a private blockchain in terms of efficiency. Indeed, public blockchains are slow and hence can process transactions only at a slow pace. In a private blockchain, as only a few nodes need to manage data, transactions can be supported and processed at a much higher pace.

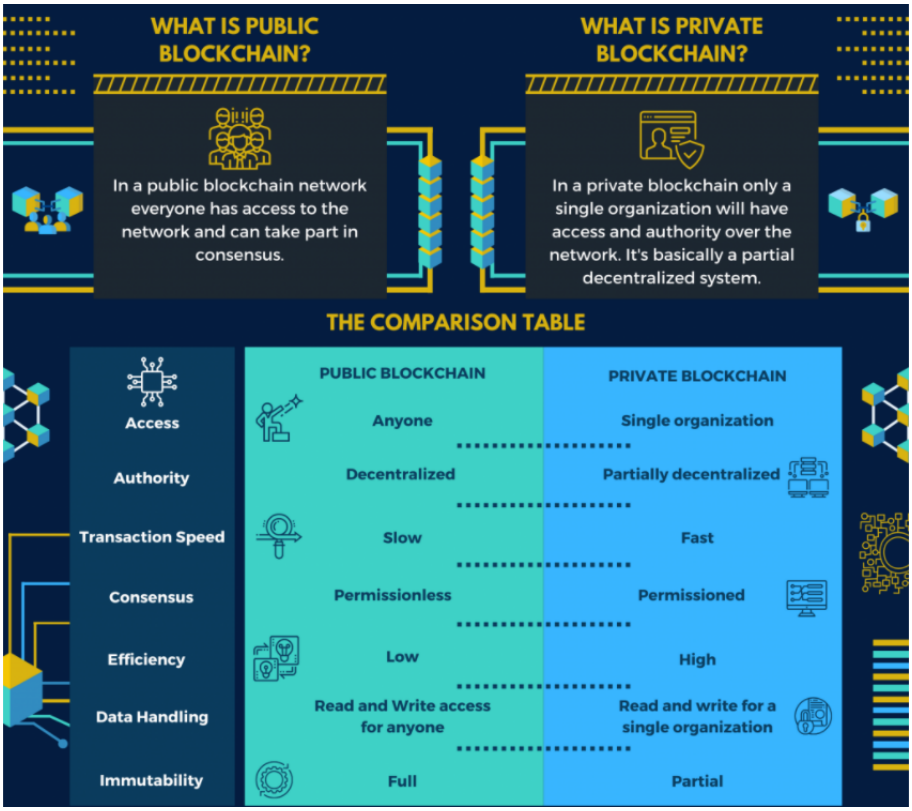


Figure 1.2.1: Public vs Private blockchains [61]

1.3 Blockchain layers

[19] The different layers of the blockchain architecture are depicted in the figure 1.3.1. The data layer stores all the transaction data and information records. The network layer enables communication between peers in the blockchain. The consensus layer includes a consensus mechanism which enables nodes to reach on the effectiveness of block data efficiently in the decentralized system where decision power is highly decentralized. The issuing of tokens is the major part of incentive mechanisms. Token issuing mechanism prescribes how tokens are generated, which would increase of total quantity of tokens. The purpose of the incentive layer is to attract participants to contribute to the computing power. The application layer focuses on developing blockchain solutions for use across different applications and industries [83].
The different layers are described in more details in the following sections.

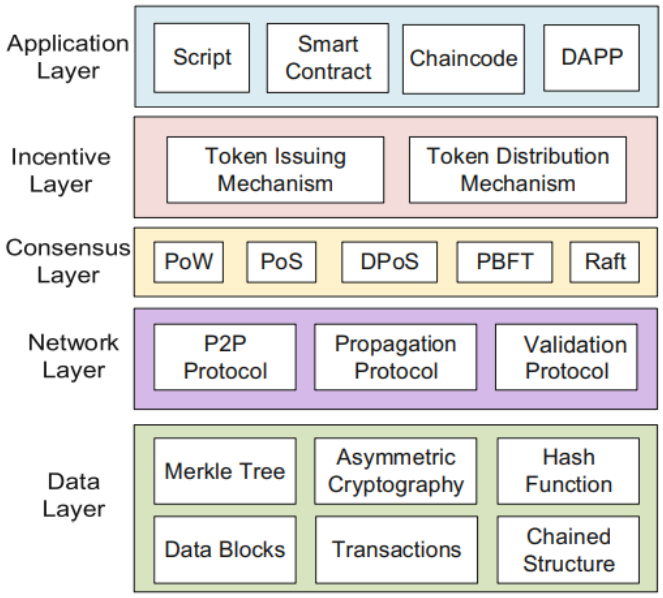


Figure 1.3.1: Overview of the blockchain architecture divided in different layers [82]

1.4 Data layer : Bitcoin case [91]

In this section, an analysis of the blockchain data layer is performed. Note that the architecture of this layer can slightly change depending on the platform that implements it. As an example, this section analyses the Bitcoin data layer.

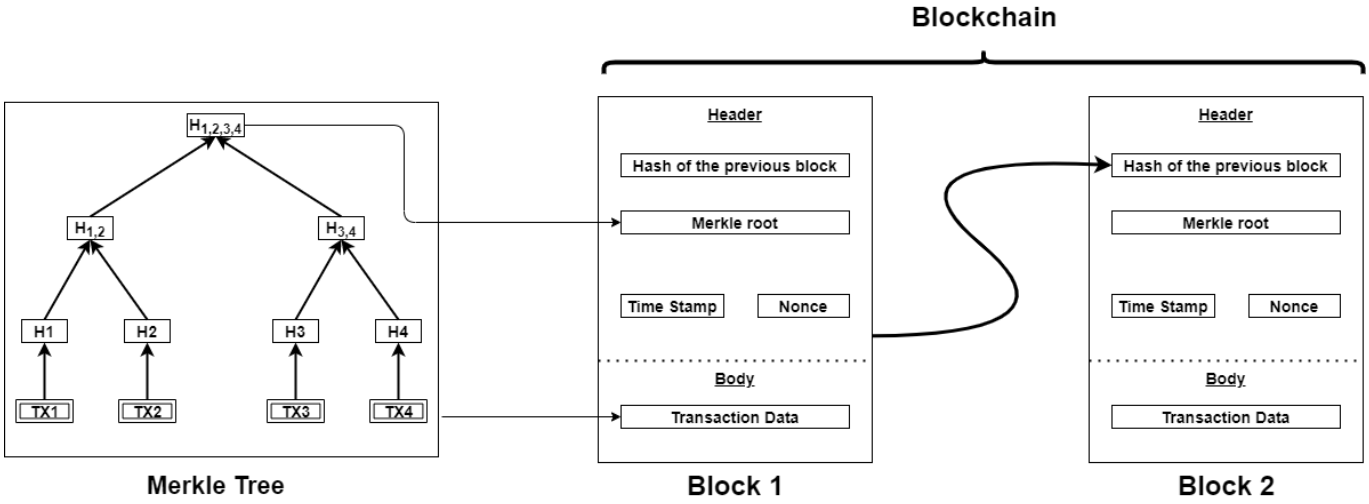


Figure 1.4.1: Architecture of the blockchain data layer implemented in the Bitcoin case

The Bitcoin blockchain is made of a succession of blocks. Each block contains a body composed of transactions and a header constituted of the hash of the previous block, the Merkle root, a timestamp and a value called Nonce. Let's go deeper in each of those components:

1.4.1 Block header

As represented in the figure 1.4.1, the blockchain header is constituted of the hash of the previous block (see section 9.2 for more information about the concept of hash), the Merkle root, the timestamp and the "Nonce" field. The timestamp is a sequence that records the time when a block is created.
In order to be valid, the block must be validated by a proof of work. As explained in more detail in the section 1.6.1, the goal of the proof of work is to find a particular sequence of characters that is added to the block such that the hash of the entire block verifies certain conditions. This sequence of characters is the "Nonce" field.
An important aspect is that the block header contains the hash of the previous block. Thanks to that, there is a link between successive blocks in the blockchain. This is a key aspect to ensure the immutability of the blockchain (see section 1.6.1).

1.4.2 Transactions and Merkle tree

As represented in the figure 1.4.1, the body of a block is made with transaction data. Indeed, as explained earlier, the role of the blockchain is to record the

transactions of the different users in a distributed ledger. Every participant can add a transaction in this ledger, which is a transfer of coin from one user to an other. According to the Bitcoin white paper of Satoshi Nakamoto [4], a coin is defined by a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership. Because the ownership of a coin is bound to a public key, the transactions are anonymous. Indeed, it is theoretically impossible to associate a public key to the identity of its owner.

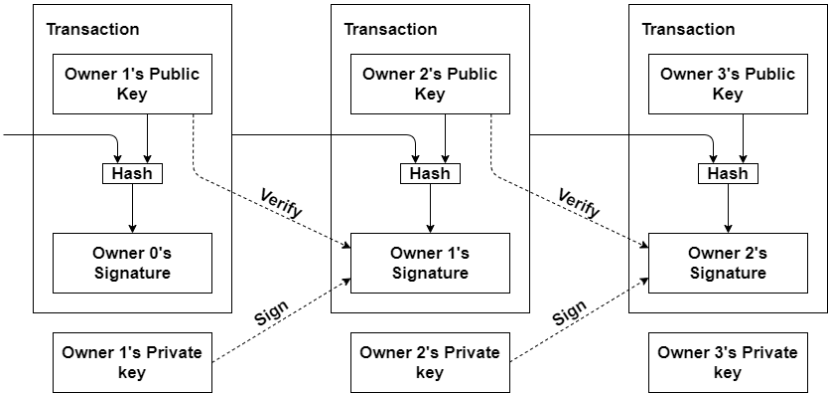


Figure 1.4.2: Representation of a coin in the Bitcoin system [4]

As represented in the figure 1.4.1 data are stored in a Merkle tree form. Indeed, as the blockchain increases, the size that it takes on each node can rapidly become a problem. In order to reduce the space that the blockchain takes in memory, some users can choose to store a lighter version of this blockchain. This is possible thanks to the Merkle tree concept.
A Merkle tree is a particular structure that stores the hash of every transaction. An example of Merkle tree is shown in figure 1.4.3. The leaf nodes of the tree are constituted with the hashes of every data (the transactions for example). The tree is build from the leafs. Each parent is computed by hashing the concatenation of its children. The last hash at the top of the tree is called the Merkle root.

1.4.3 Merkle proof [9]

A characteristic of the Merkle tree is that it eases the authentication of data. The figure 1.4.3 depicts an example of authentication. In this example, the verifier has the authentic h_r (Merkle root) and wants to verify the integrity of x_2 and x_7. The prover then provides the verifier with this information: x_2, x_7, $h(x_1)$,

h_d, $h(x_8)$ and h_e. The verifier can then verify x_2 and x_7 by first computing $h(x_2)$, $h(x_7)$, $h_c = h(h(x_1)||h(x_2))$, $h_f = h(h(x_7)||h(x_8))$, $h_a = h(h(x_c)||h(x_d))$, $h_b = h(h(x_e)||h(x_f))$ and $h_r = h(h(x_a)||h(x_b))$ and then checking if the calculated h_r is the same as the authentic one.

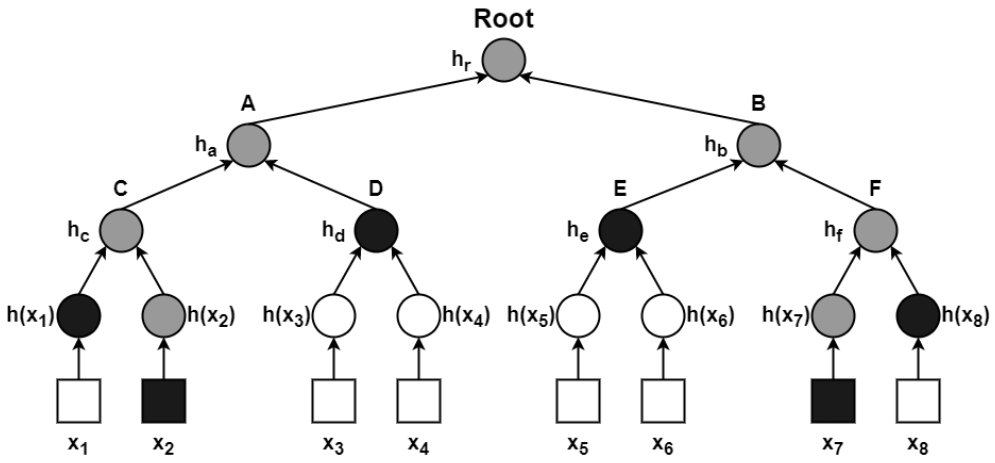
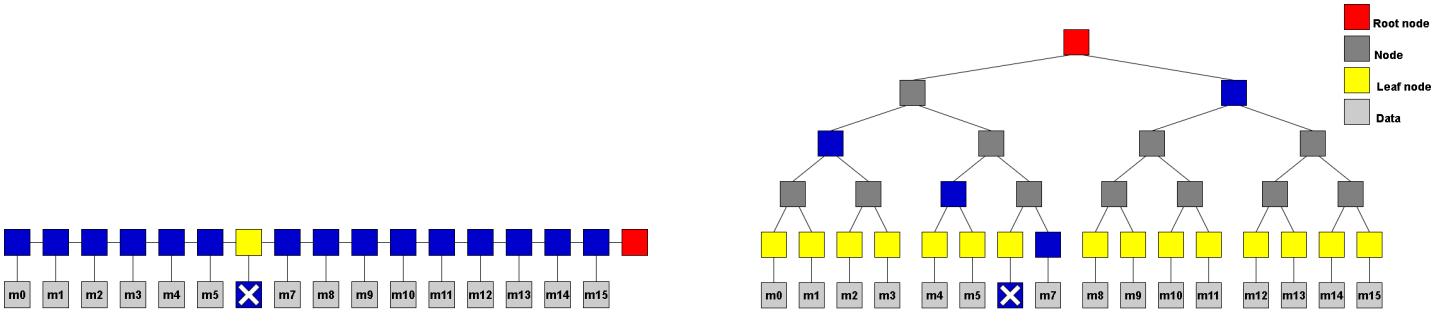


Figure 1.4.3: Merkle proof [9]

1.4.4 Advantages of the Merkle tree

The main advantage of this structure is that less nodes have to be compared to proof that a node is tampered with. Indeed, an other solution would be to compute the hash of all the messages, to append all those hashes in a string and to compute the hash of this string (cf. figure 1.4.4 a)). However to show that data m6 is tampered with, 16 nodes are needed (nodes in blue in the figure 1.4.4 a)). With the Merkel tree implementation, only 5 nodes are needed (nodes in blue in the figure 1.4.4b)). (see section 1.4.3 for an example of the authentication process).



a) Global hash b) Merkle tree

Figure 1.4.4

The second advantage of the Merkle tree is that it permits the users who do not want to store the entire blockchain to store only a lighter version of it. In this case, the user does not store all transactions corresponding to a block but it only stores the Merkle root. If the user wants to verify the correctness of a transaction stored on a particular block, it will request the Merkle tree of that particular block, recompute the Merkle tree of that block, and verify that the computed Merkle root corresponds to the Merkle root that the user stores on its lightweight blockchain.

1.5 Network layer

The network layer includes the three protocols used in the blockchain network. Its role is to enable communication between peers in the blockchain.
The **Peer-to-Peer (P2P) protocol** is used to construct the blockchain network. This protocol includes the peer discovery and the connection to the other peers when the blockchain starts. Moreover, it includes a mechanism of synchonization of local data between peers.
The **propagation protocol** is used to broadcast transaction proposals and new executed transactions which is important for the creation of a new block.
The **validation protocol** is used to validate the transactions before being committed to the ledger.

1.6 Consensus layer

The consensus is a key concept in the blockchain conception. Indeed, the consensus is a fault-tolerant mechanism that is used to achieve the necessary agreement on a single data value or a single state of the network among distributed processes. Thanks to it, the distributed nodes record the same information in their

distributed ledger. There are several kinds of consensus. The most popular one are explained in this section.

1.6.1 Proof of Work (PoW)

One of the most popular consensus is the proof of work. This consensus is used for public blockchains such as Bitcoin or Ethereum. The goal of the proof of work is to define an expensive computer calculation, also called mining, that needs to be performed in order to create a block. This expensive computer calculation is often linked to a hash problem. The most common one is to find a random serie of characters (called nonce) such that the hash of those characters added to the content of the block gives a result with specific characteristics (for example the result of the hash begins with 30 zeros). The only way of finding this particular chain of character is to try all the possibilities. Once a first miner found a solution to the problem, the other nodes verify that the proposition is a correct solution to the mathematical problem. This verification is less power consuming than the mining. Indeed, verifying that a particular serie of characters added to the block verifies a condition only requires one call to the hash function. Much more call to the hash function are needed in order to find them. If the other nodes validate the block, it is added to the blockchain. The node who mined it receives a reward and some transaction fees payed from transaction senders.
Other miners accept the newly generated block by starting to work on the consecutive block. Crucially, all succeeding blocks contain hash outputs from all preceding blocks. As generation of hash outputs is random and performed in parallel by many miners, multiple chains may appear. In this occasion, the network stores all resulting chains. Network members eventually abandon all other chains except the longest, which is assumed to have been produced by a network majority of computational power and therefore to represent the most valid state of the ledger. As a result, malicious attackers are constantly outpaced by the honest part of the network (see the two examples in the 'conflict and security' part of this section), unless they can control more than 51% of the computational power in the network. In the case of a 51% attack, malicious nodes could potentially rewrite all history of transactions.
Nodes need to store multiple chains when they appear, however the older a block in the chain, the more unlikely it is for it to be reversed. A typical number of confirmations accepted by the majority of the Bitcoin community and wallet providers is 6 confirmations. It takes approximately 1 hour for the block to get accepted since one block is mined every 10 minutes on average. This timing is maintained constant thanks to the balance between the total network power and the difficulty to find a nonce with specific characteristics; the difficulty increases with the expansion of the network.

The major drawback of the proof of work is that it consumes a lot of power since a lot of miners try to resolve a complex problem at the same time. According to Cambridge researchers [46], the Bitcoin mining process consumes around 121.36 terawatt-hours (TWh) which is more than the electric consumption of Argentina (121TWh).

Conflict and security

Let's analyse the behavior of the proof of work in case someone wants to create false transactions. Imagine that the figure 1.6.1 represents the blockchain stored by Bob. In this context, Alice creates a block that contains a transaction that she only broadcasts to Bob. Because this block is only transmitted to Bob, the other miners continue to mine from the basis of the block 2. If Alice wants Bob to trust her wrong block, she must continue mining the next blocks from the basis of her wrong block. Moreover, she must mine faster than all the other miners together. Indeed, Bob will take for granted the blockchain that has the highest number of blocks. Alice will not be able to mine faster than the other miners together unless she has more than 50% of the computing resources of all the miners. At the end, her wrong block will so be rejected by Bob.

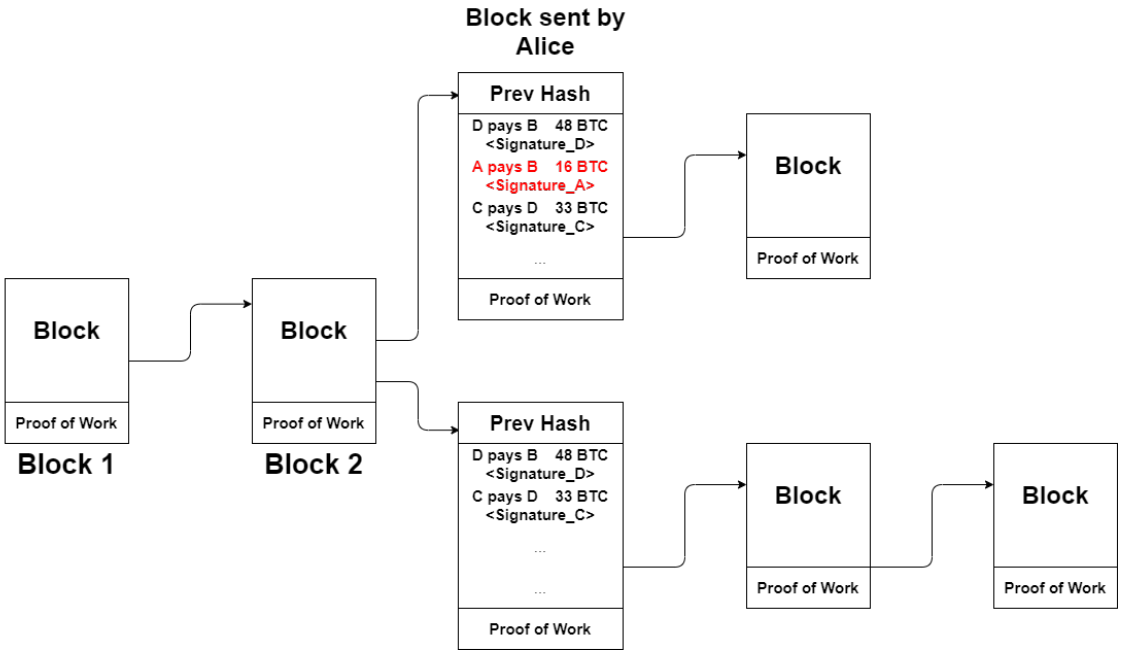


Figure 1.6.1: Transaction sent to a single user

Let's have a look to an other scenario. In the figure 1.6.2, Alice wants to reduce the amount of money that she gave to Bob. She is able to do this operation since she has the private key that signed the transaction. However, the change of content of one block will modify the PoW and the hash of this block. Moreover, because the blocks contain the hash of their previous block, changing the PoW of the block 2 will change the PoW of all the next blocks. The PoW of the block 3 will so be incorrect and so on. The others miners continue to mine from the basis of the correct block 4. If Alice wants her modification to be accepted by the other (and so produce the blockchain with the highest number of valid blocks), she will again have to mine faster than all the other miners together. Again, she will not be able to do that unless she has more than 50% of the computing resources of all the miners. In summary, two versions of the blockchain can exist but only the longest one is considered as valid. For that reason and as explained earlier, people consider that a block is completely valid if 6 blocks have been created after it.

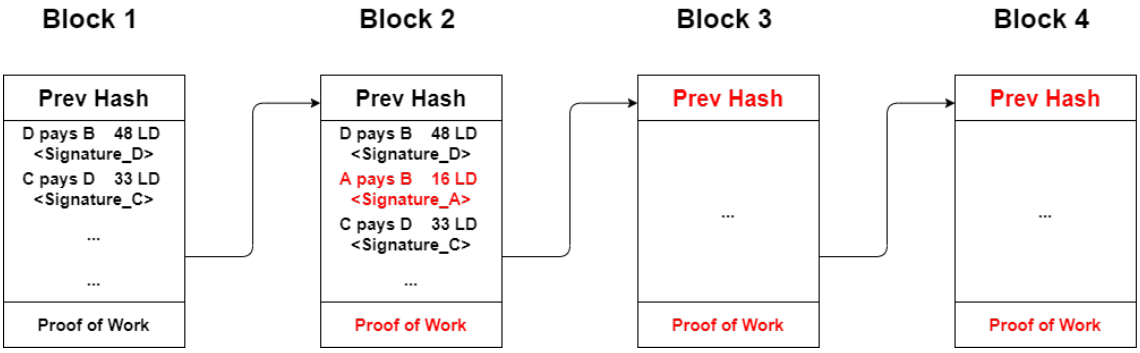


Figure 1.6.2: Transaction modified by a user

A characteristic of the Bitcoin is that it is the first practical implementation of a P2P network that managed to avoid the double-spending problem [70]. As the term suggests, double-spending means spending the same money twice. With physical cash such as coins and notes, this is simply not possible and therefore it is not an issue. Indeed, imagine you go to a supermarket and pay 2€ for a coffee. As soon as you give the money to the cashier, you can't re-spend it unless you physically steal it. However, digital money is different from cash. When someone makes a transaction with digital cash, this transaction is broadcasted to all the 'nodes' in the network. These nodes need to receive and confirm the transaction, which takes time. Double-spending would occur if during this time, the transaction is copied and rebroadcasted before it has been confirmed on the network.
An example of double spend is represented in the figure 1.6.3, 1.6.4 a) and 1.6.4 b). At the beginning, Alice only has 15 ₿ on her wallet. B sells its music at 15 ₿ and D sells its movie 15 ₿. As explained in the section 1.4.2, a coin is defined by a

chain of digital signatures. A double spend situation would occur if Alice signs a transaction of 15 ₿ from her to B and signs a transaction of the same 15 ₿ from her to D (see figure 1.6.4 a). If B and D are not careful, they will respectively send the music and the movie to Alice. At the end, they will both think that Alice sent them 15 ₿. However their ledger are now different (see figure 1.6.4 b)). After some other blocks have been created, B and D will take for granted the chain with the most blocks in it. At this moment, B and D will know that Alice fooled one of them.

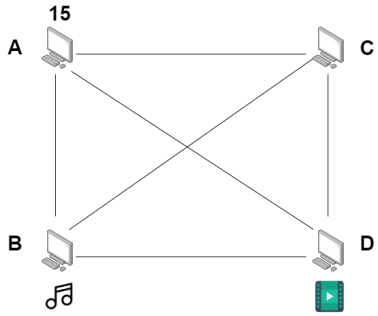
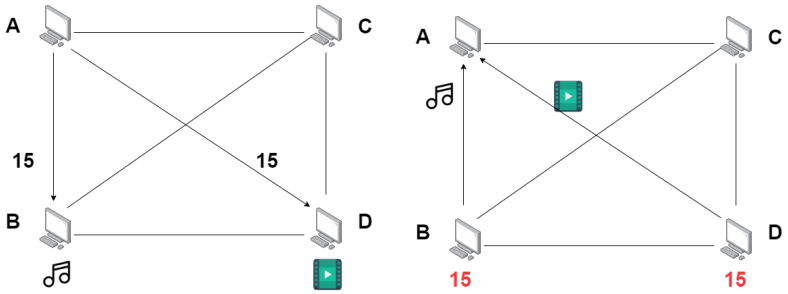


Figure 1.6.3: Example of double spending



a) b)

Figure 1.6.4

This double spending scenario is also represented in term of transaction in the figure 1.6.5. In this figure, it can be observed that even if A signed the transaction to B, nothing prevents A from resigning the same transaction to D.

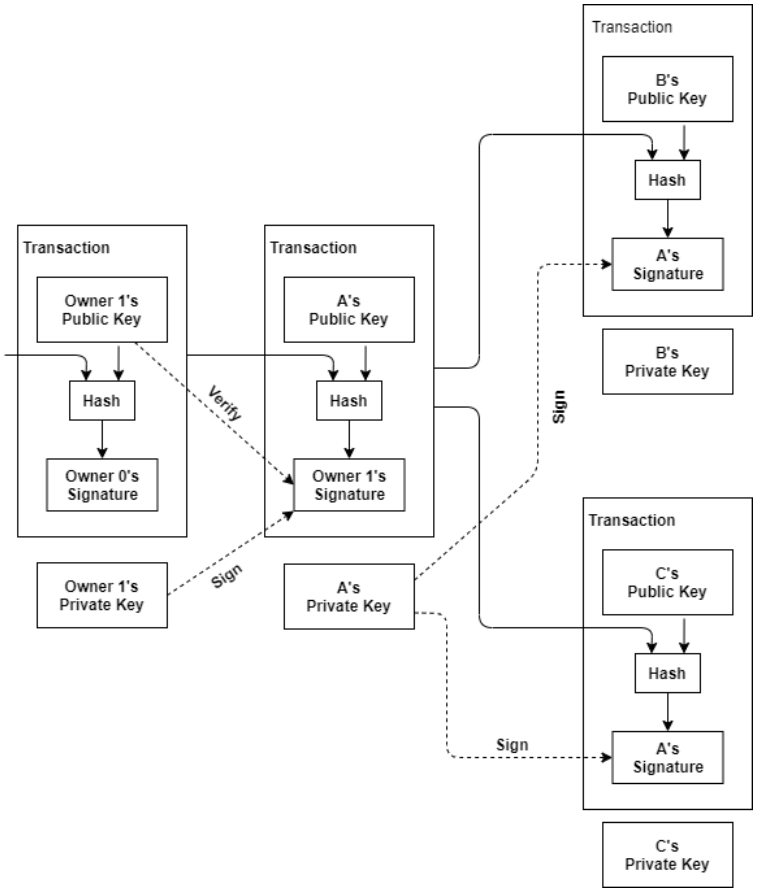


Figure 1.6.5: Double spending represented as sign transaction

A good consensus protocol guarantees that this behavior is impossible. Indeed, imagine that Alice only has 15 ₿ on her wallet and tries to give it first to D and then tries to give the same 15 ₿ to B. If the transaction from A to D is the first to be accepted by the network, then this transaction will be put on a block and the other nodes will construct the rest of the blockchain from the basis of this block. Because the longest chain of block is considered valid by the other nodes, the transaction from A to B will be rejected by the network. Indeed, from their point of view, Alice paid 15 ₿ to D and she has no remaining currencies on her wallet.
It is possible that a block that contains the transaction from A to B is validated by the network at the same time that an other block that contains the transaction from A to D. In this case a fork is created. Nonetheless, one of the two block will finally be contained in the longest chain of block and the other will be dropped. That is the reason why Bitcoin users wait for 6 other blocks to be created before considering a transaction to be completely valid.

1.6.2 Proof of Stake (PoS) [44]

Criticism to the huge amount of energy used by PoW led to alternative algorithms such as proof of stake (PoS). In the early version of PoS, people only needed to own tokens (currency unit) in order to be eligible to be a validator (equivalent to a miner in the PoW). The role of the validator is then to forge (equivalent to minning in the PoW) the next block. After forging it, other validators verify the created block. If it is correct, the producer of the block earns the transaction fees that are paid for using the network.
The idea behind the PoS concept is that if a person owns a PoS network's token, it has an interest in the success of that network. The more tokens it owns, the more it has "at stake" if the network is attacked. Indeed, if the network is successfully attacked, the value of the tokens is likely to significantly drop.
Under this logic, a person who owns a lot of token is more likely to be eligible to forge the next block. Indeed, this person has more to lose if it disrupts the network.

The biggest advantage of the proof of stake compared to the proof of work is that it does not require large amount of electricity neither expensive investment in hardware. Indeed, there is no complex computation to execute. An other advantage is that PoS can handle more transactions per second than proof of work.

However, PoS also has drawbacks. When PoS first appeared, a large part of the crypto community was unconvinced that merely owning a token was enough to disincentivize bad behavior. One of their major concerns is called the nothing at stake problem. This vulnerability appears when someone loses nothing when behaving badly, but stands to gain everything. Imagine that a fork appears in the blockchain (see figure 1.6.6).

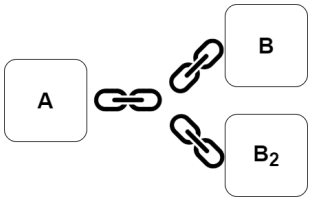


Figure 1.6.6: Fork in the blockchain

Now, participants have to pick one of the two chains. Ideally, one chain will be picked (see figure 1.6.7).

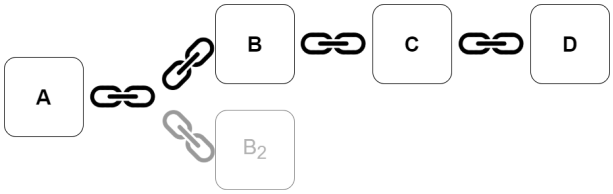


Figure 1.6.7: Ideal case when there is a fork

However, a problem can happen. Imagine the case of the figure 1.6.6. Instead of choosing the good one, C could also produce its block and put it on both chain. Indeed, if C picks only one chain, it risks losing the transaction fees on the orphaned chain. However, if C picks both chains, he simply has to wait for one of the chains to be picked as winner and he gets his rewards either way. C can forge on multiple branches at the same time because there is no resource being used to create a block. At first glance, it may not seem like a big deal. However, it can compromises the security of the blockchain. Indeed, this situation could lead to double spend problem. Imagine that there is a fork and all validators are actively building on both forks. Flo, the attacker, is interested in executing a double spend attack. He buys some bitcoin with its cryptocurrency on one fork and on the other fork, pays a bill with the same cryptocurrency. After that, Flo sells its Bitcoin and receives physical money. Those blocks have been forged by 'C' on both forks. Flo then just has to build a block only on the second fork. Now that there is a clear longest chain, the entire network converges on the second fork. Flo just performed a so call 'double spending' attack [64]. This scenario is represented in the figure 1.6.8.

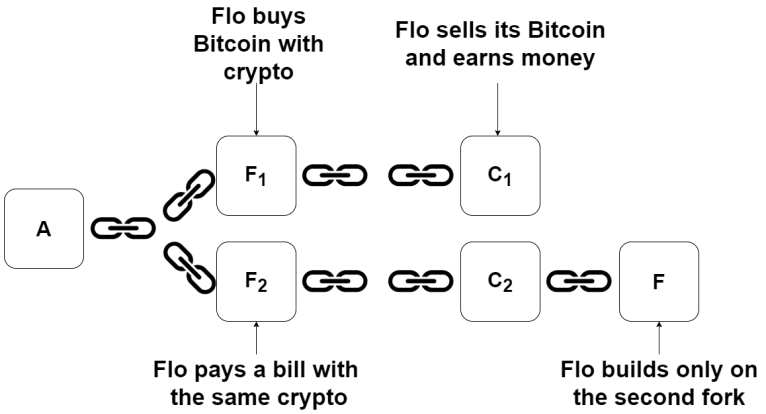


Figure 1.6.8: Double spend in the fork situation

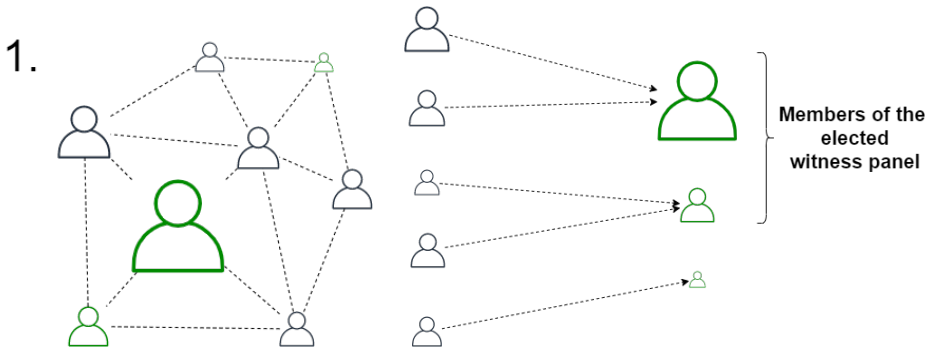
It is important to note that for this example, it is assumed that all participants act "optimally" ('C' forged on both chains). In a more realistic scenario, some honest validators could refuse to build on forked chains. However, the nothing at

stake problem led researchers to propose some variants of the PoS that are still today under analysis. For example, people suggested that the validator has to deposit part of its coins as a stake before creating a block. If the validator validates fraudulent transactions, it is likely to lose this stake. As long as the stake is higher than the reward associated with the validation of a transaction, a validator is assumed to correctly do his job. If it does not correctly do its job, the validator loses more money than it gains.
An other drawback is that people with a large amount of coins can have an outsized influence on the network and make it more centralized. That is the reason why the 'coin age' selection method tries to prevent this. In this method, part of the validator selection process depends on the time that a user waits to create a block. The longer it has already waited, the more chance it has to become to next validator to produce a block. However, the size of the stake is always a factor in selecting validators. So this problem can not be entirely avoided.

1.6.3 Delegated Proof of Stake (DPoS) [21] [51]

Delegated Proof of Stake is mostly maintained through the election process. Active users of DPoS-based blockchain are voting for "witnesses" and "delegates" with placing their tokens on the name of their candidate (those tokens are not spent this way, they are just representing the position of stakeholder and remain his/her property). Positions of the witnesses and delegates differ in various cryptocurrencies and one role can absorb another role's functions or even eliminate it. In case of BitShares (first DPoS-based blockchain), witnesses are responsible for creating and validating blocks. Top-tier witnesses are awarded with fees for every validated transaction. Since it is the witnesses responsibility to validate transactions and produce blocks, it is important they have a stable server close to 100% up-time.
On the other hand, delegates are voted to govern the system and to propose core changes. Delegates are not in charge of block production and transaction validation, but they oversee such parameters as transaction fees, block sizes, witness pay, and block intervals of the network.
Election voting is a continuous process so every witness or delegate is under the pressure of losing its place to another, more popular competitor. Reputational and financial losses are primal motivation for their abstinence from malicious behavior. Most DPoS-based blockchains take stakeholder's stake size into account. Vote power of the voter is determined by the amount of tokens he is holding. However, there is no regulation that restricts users from voting due to their stake being not big enough.
The figures 1.6.9 a) and 1.6.9 b) illustrate the operation of the DPoS. In the first phase, nodes express interest in becoming a witness and begin lobbying, making positive contribution for the network and engaging the community.

In the second phase, people in the network allocate their tokens as votes for witnesses. The more tokens they have, the higher their voting weight. Note that participants are not giving tokens to their witnesses. They are merely allotting funds to their choices as an expression of their vote. They can reassign their tokens to another witness at any time.
Finally, there is a ranking of nodes with the most votes. The top N of these will become members of the elected witness panel. N depends on the network.



a) First phase b) Second phase

Figure 1.6.9

Here are the advantages of the DPoS:

- DPoS is much more scalable than PoW as it never starts requiring high computing power and is generally approachable for users with poor equipment.
- DPoS blockchains showed themselves to be faster than PoW and PoS-based blockchains.
- DPoS is energy efficient and environmentally friendly.

However here are the major drawbacks:

- DPoS systems are vulnerable to centralization as a number of witnesses is strictly limited.
- DPoS blockchain is exposed to flaws of classic real-life voting. For example, DPoS users with small stakes can decide that their vote doesn't matter in comparison with votes of bigger stakeholders.

1.6.4 Proof of Authority (PoA) [74]

Proof of Authority is a modified form of Proof of Stake where instead of stake with the monetary value, a validator's identity performs the role of stake. In this context, staking identity means voluntarily disclosing who you are in exchange for the right to validate the blocks. This means that the benefits you derive from it are public and so are the nefarious actions you might undertake. A few conditions need to be satisfied for the concept to work:

- Identity must be true: meaning there needs to be a standard and robust process of verifying that validators are indeed who they claim they are.
- Eligibility for staking identity should be difficult to obtain: so that the right to be a validator becomes earned, valued, and unpleasant to lose.

PoA is suited to private blockchains in which authority node identity may be established and disclosed within the private network [73].
The advantages of PoA are:

- Proof-of-Authority based networks are of very low demand on the computing power of user.
- Transaction time of PoA is significantly faster than the transaction time of PoW-based networks.
- PoA networks are very scalable.

The drawbacks of the PoA are:

- PoA based networks strongly lack decentralization. In contrast to PoW, PoA blockchain can have only a limited number of validators.
- The risk of damaging the reputation does not necessarily keeps a person from participating in malicious actions. Indeed, the size of the gains that can be gathered with a reputation-destroying event can be more valuable than the reputation in community.

1.6.5 Proof of Elapsed Time (PoET) [48] [28]

PoET was developed by chip manufacturing giant Intel back in 2016 as an efficient consensus mechanism primarily for permissioned blockchain networks. The PoET is based on the concept of trusted execution environment (TEE). In 2015, Intel produced such a TEE called SGX. SGX is a sophisticated technology, but

at its core, it is a set of instructions for a CPU that is used by applications to isolate specific, trusted regions of code and data. It provides a secure enclave for developers to protect sensitive data or code from outside interference or inspection. Code that runs in a TEE using SGX can produce a signed attestation from within a platform or application that is rooted in the processor and provides authentication that the code has been correctly initialized in a trusted environment.
In the PoET, each network participant is given a random timer object and the first timer to expire "wakes up" that participant who becomes the block leader and produces the new block. Here is how it works: in the initial phase of the consensus, a potential participant in the network downloads the trusted code and propagates a "join" message to the network with the signed attestation from SGX, which has produced a public/private key pair. The network of nodes then either accepts or rejects the attestation. If accepted, the participant joins the network and can participate in the elapsed time, randomized lottery selection process. The second phase of the consensus mechanism is the elapsed time lottery random selection. In each round of consensus, network participants receive a signed timer object from the trusted code which is completely randomized. The idea is to mitigate any potential malicious actor from gaming the system and attempting to consistently receive a shorter timer so that they can produce more blocks. Each participant subsequently waits for their randomized timer to expire. The network participant's timer that is the first to expire propagates a signed certificate to the network indicating that they are the randomized block leader for that round. The message is authenticated, and the block is produced. The round then restarts.
Thanks to reliable devices, PoET eliminates the requirement of expensive mining in PoW.

1.6.6 Practical Byzantine Fault Tolerance (PBFT) [28]

This algorithm has been designed to solve issues of Byzantine Fault Tolerance solutions (see section 9.3 for more information about the different type of failures) and to work effectively in asynchronous systems. For this algorithm, nodes are first sequentially ordered. One node is the leader node (or the primary node) and others are backup nodes also called secondary nodes. In the case of a primary node failure, any eligible secondary node can be assigned as the primary node. PBFT works on the principle that the number of malicious nodes should not be greater than or equal to one third of the total nodes in the system. If required, honest nodes can vote and replace the current leader node by the next node in the line.
All nodes are constantly involved in the consensus. Due to this characteristic, the transactions can be agreed upon and finalized without needing multiple confirmations [72] [55]. At the contrary of the Bitcoin (where 6 more blocks are needed in order to confirm a transaction), there is no waiting period to ensure a transaction

is secured after including it in a block [90]. This algorithm is energy efficient. The disadvantage of PBFT is that it is prone to Sybil attacks and scaling issues.
The Sybil Attack is a type of attack seen in peer-to-peer networks in which a node in the network operates multiple identities actively at the same time and undermines the authority/power in reputation systems. The main aim of this attack is to gain the majority of influence in the network to carry out illegal (with respect to rules and laws set in the network) actions in the system. In this attack, a single entity has the capability to create and operate multiple identities (user accounts, IP address based accounts). To outside observers, these multiple fake identities appear to be real unique identities [56]. Note that the 51% attack in the case of the proof of work is a particular type of Sybil attack. However, this attack is more complicated to perform in the case of a PoW. Indeed, in this case, the attacker has to own more than 51% of the total computational power in the network which is very expensive. The Sybil attack is less costly in the case of the PBFT attack. Indeed, this algorithm does not require expensive investments for a node to participate in the consensus.
An other drawback of the PBFT is that it does not scale well because as represented in the figure 6.2.14, each node has a link and communicates with all the other nodes of the network.

The PBFT is described with more details in the section 6.2.7.

1.6.7 Conclusion

Consensus Algorithm	Blockchain Platform	Pros	Cons
Proof of work (PoW)	Bitcoin, Ethereum	-High scalability -High security for a large number of nodes	-High energy consumption
Proof of Stake (PoS)	Ethereum, NavCoin	-Less amount of hardware -Fast	-Nothing at stake attack -The rich get richer
Delegated Proof of Stake (DPoS)	Bitshares, EOS	-Fast -Efficient	-More centralized
Proof of Authority (PoA)	VeChain	-Fast -Scalable	-More centralized
Proof of Elapsed Time (PoET)	Hyperledger-Sawtooth	-Fair -Efficient -Scalable	-Dependent on Intel technology
Practical Byzantine Fault Tolerance (PBFT)	Hyperledger-Sawtooth	-Require less number of confirmation -Energy efficient -Fast	-Less scalable -Prone to Sybil attack

Note : Ethereum is currently working with the PoW algorithm but it is working towards replacing its PoW mechanisms with PoS. Note also that PoS, DPoS, PoA, PoET and PBFT were created after the PoW and all have the advantage of consuming less energy than the PoW.

1.7 Incentive layer [82] , [29]

The issuing of tokens is the major part of incentive mechanism. Token issuing mechanism prescribes how the token is generated, which would increase of total quantity of tokens. For example, in the Bitcoin process, where the PoW consensus is adopted, the creation of a block takes time and is resource consuming. In order to encourage node participation, the blockchain rewards miners in the form of virtual currency. In the Bitcoin case, the reward associated for mining a block is reduced by two every times the blockchain gains 210000 blocks. For example, between January 2009 and November 2012, the reward was 50 ₿ per mined block while it dropped to 25 ₿ per block between November 2012 and July 2016. Nowadays, a reward of 12.5 ₿ is given per mined block.

1.8 Application layer

Thanks to the application layer, developers can design their business application using the underlying blockchain as the database. Such business logic is implemented using the programming mechanism provided by blockchain systems such as smart contract [82].

1.8.1 Smart contract [33]

Blockchain was initially designed for P2P money only. Nonetheless, it soon showed the potential to be used for any kind of P2P value transaction on top of the internet. The Ethereum project thus introduced the idea of dedicating a specific contract layer to blockchains, where the ledger itself is used by smart contracts that trigger transactions automatically when certain pre-defined conditions are met.
The concept of smart contract was firstly introduced by Nick Szabo in 1997 (before the concept of blockchain) in his article named : "Smart Contract" [85] [23]. The definition is the following :
"A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitration and enforcement costs, and other transaction costs".
Smart contract can be seen as digitalized contract; they are pieces of code running on top of a blockchain network, where digital assets are controlled by that piece of code implementing arbitrary rules. They have properties of contractual agreements. If and when all parties to the smart contract fulfil the pre-defined arbitrary rules, the smart contract will automatically execute the transaction. Smart contracts can be used for simple economic transactions like sending money from A to B. They can also be used for registering any kind of ownership and property rights.
Let's take an example. Imagine a crowdfunding platform where investors place money to support a project. If the project collects enough money, the project team expects to receive the money from the crowdfunding platform. On the other hand, investors expect to be refund if the project did not collect enough funds. In this example, the crowdfunding platform can be seen as a third party in which the project team and the investors have to trust. To avoid breach of contract, this third party could be replaced by a smart contract. The smart contract will be a piece of code programmed such that it holds the received funds until a certain goal is reached. If this goal is reached, the smart contract automatically transfers the money to the project creator. If the project fails to meet the goal, then the money automatically goes back to the supporters. In this case, because the contract is

written as a piece of code, all the process is automatic. The construction and execution of smart contracts based on blockchain include the following steps:

1. Construction of the smart contract: multiple users in the blockchain participate in formulating the smart contract.

2. Storage of the smart contract: smart contract spreads to each node through the P2P network and is stored in the blockchain.

3. Execution of the smart contract: the smart contract regularly checks the status of the automaton to verify transactions that meet the conditions. When all the conditions of the smart contract are met, it automatically executes. [34].

Because smart contracts are stored on a blockchain, they inherit some interesting properties. The first one is that they are immutable. In this case, when the code has been created, it can never be changed. Nobody can tamper with the code and change the term of the contract. An other property is that it is distributed which means that a single person can not cheat and force the contract to release the funds. Indeed other people in the network would spot this attempt and mark it as invalid. Here are the main advantages of smart contracts:

- Immutable and secure: as explained earlier, once the contract is written, there is no way to modify it without the consent of the two parties.

- Transparency: the terms and conditions of these contracts are fully visible and accessible to all relevant parties. There is no way to dispute them once the contract is established.

- Speed: these contracts run on software code and live on the internet. As a result, they can execute transactions very quickly. This speed can shave hours off many traditional business processes since there is no need to process documents manually.

- Storage and Backup: all the documents stored on blockchain are duplicated multiple times; thus, originals can be restored in the event of any data loss. This is due to the decentralized characteristic of the blockchain.

However, smart contracts are far from perfect and some disadvantages can also be point out. One of the biggest problem is that the code of the smart contract must have no bugs or faults in it. In 2018, Ivica N. et al [17] analysed nearly one million of smart contracts. They found around 3.4% of them to be faulty by only checking via an algorithm for the most common exploit possibilities. This represents hundreds of millions of dollars being at risk of being stolen or frozen.

Such attacks already happened in the past. In 2016, the DAO (a decentralized venture capital fund where investments were based on votings by the community) was attacked by unknown hacker, exploiting a combination of vulnerabilities in the DAO smart contract. The hacker was able to obtain 3.6 million ether (digital currency of the Ethereum network) which was valued around $50 million USD at the time of the attack [67].
Let's also point that although there is no need of third party in the process of fulfillment of Smart Contracts, they will assume different roles from the ones they take in traditional contracts. For example, lawyers are still needed to work with developers to understand the terms and create codes for smart contracts.

Chapter 2

Microgrids [7]

This chapter introduces the concept of the microgrid. This introduction will help to understand the next chapter which explains the benefits that can be achieved thanks to the combination of microgrids and blockchains.

2.1 Glossary of the abbreviations

Here is a glossary of abbreviations used in this chapter.

- CHP: Combined heat and power
- DER: Distributed energy resources
- DG: Distributed generation
- DSO: Distribution system operator
- ESS: Energy storage system
- GHG: Green house gas
- LV: Low-Voltage
- MC: Microsource controller
- MG: Microgrid
- MGCC: Microgrid central controller
- MV : Medium voltage
- RES: Renewable energy source
- PV: Photo voltaics

2.2 Introduction

In recent years, microgrids (MGs) have become a research hotspot in the energy field. At present, there are more than 400 microgrid demonstration projects in the world under planning, constructed and put into use [35].
Here is the definition from the EU research projects:

"Microgrids comprise Low-Voltage (LV) distribution systems with distributed energy resources (DERs) (microturbines, fuel cells, PhotoVoltaics (PV), etc.), storage devices (flywheels, batteries) energy storage system and flexible loads. Such systems can be operated in both non-autonomous way (if interconnected to the grid) or in an autonomous way (if disconnected from the main grid). The operation of microsources in the network can provide distinct benefits to the overall system performance, if managed and coordinated efficiently. " [84]

Here are major messages delivered from this definition:

1. Microgrid is an integration platform for supply-side (microgeneration), storage units and demand resources located in a local distribution grid.
    - In the microgrid concept, there is a focus on local supply of electricity to nearby loads, thus aggregator models that disregard physical locations of generators and loads (such as virtual power plants with cross-regional setups) are not microgrids.
    - A microgrid is typically located at the LV level with total installed microgeneration capacity below the MW range, although there can be exceptions: parts of the medium voltage (MV) network can belong to a microgrid for interconnection purposes.
2. A microgrid should be capable of handling both normal state (grid-connected) and emergency state (islanded) operation. If a microgrid exchanges power with the distribution grid, it operates in grid-connected mode. In islanding or stand-alone mode, power is exchanged only locally between loads, DERs and ESSs (energy storage system) [13].
    - The majority of future microgrids will be operated for most of the time under grid-connection, except for those built on physical islands. The main benefits of the microgrid concept will arise from grid-connected (i.e. “normal”) operating states.
    - In order to achieve long-term islanded operation, a microgrid has to satisfy high requirements on storage size to continuously supply all loads or it has to rely on significant demand flexibility.

2.3 Operation strategies of microgrids

Depending on the stakeholders involved in the planning or operation process, four different microgrid operational objectives can be identified: economic option, technical option, environmental option and combined objectives option (see figure 2.3.1).
In the economic option, the objective function is to minimize total costs regardless of network impact/performance. This option may be envisaged by distributed generation (DG) owners or operators. DGs are operated without concern for grid or emission obligations. The main limitations come from the physical constraints of DG.
The technical option optimizes network operation (minimizing power losses and voltage variation), without consideration of DG production costs and revenues. This option might be preferred by system operators.
The environmental option dispatches DG units with lower specific emission levels with higher priority, disregarding financial or technical aspects. This is preferred for meeting environmental targets, currently mainly supported by regulatory schemes. DG dispatch is solely determined by emission quota; only DG physical limitations are considered.
The combined objective option solves a multi-objective DG optimal dispatch problem, taking into account all economic, technical and environmental factors. It converts technical and environmental criteria into economic equivalents, considering constraints from both network and DG physical limits. This approach could be relevant, for instance, for actors that participate not only in classical energy markets, but also in other potential markets for provision of network services and emission certificates.

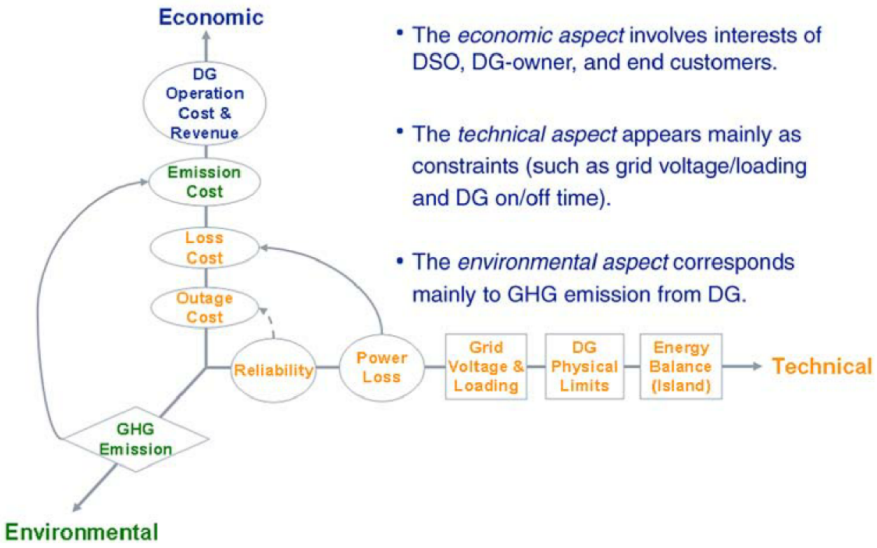


Figure 2.3.1: Microgrid operation strategies [7]

2.4 Centralized vs decentralized

The microgrid structure can be operated in a centralized or decentralized way, depending on the responsibilities assumed by the different control levels. In centralized control, the main responsibility for the maximization of the microgrid value and the optimization of its operation lies with the MGCC (microgrid central controller). The MGCC using market prices of electricity and gas costs, and based on grid security concerns and ancillary services requests by the distributed system operator (DSO), determines the amount of power that the microgrid should import from the upstream distribution system, optimizing local production or consumption capabilities. The defined optimized operating scenario is realized by controlling the microsources and controllable loads within the microgrid by sending control signals to the field.
In a fully decentralized approach, the main responsibility is given to the microsource controllers (MCs) that compete or collaborate, to optimize their production, in order to satisfy the demand and probably provide the maximum possible export to the grid, taking into account current market prices. This approach is suitable in cases of different ownership of DERs, where several decisions should be taken locally, making centralized control very difficult.
The choice between the centralized and decentralized approach for microgrid control, depends on the main objectives and the special characteristics of the controlled microgrid and the available or affordable resources: personnel and equipment. Clearly, centralized control is more suitable if the users of the microgrid (DG and

load owners) have common goals or a common operational environment seeking cooperation, in order to meet their goals. Such an example is an industrial microgrid, in which a single owner might exercise full control of all its energy sources and loads, is able to continuously monitor them and aims to operate the system in the most economical way. In this case, the number of nodes is generally limited and it is relatively easy to install a fast communication system and a set of sensors. Dedicated operating personnel for the operation of the microgrid might be available. Furthermore, the optimization problem has a limited set of constraints and specific objectives, such as cost minimization. Finally, the requested solution should be as accurate as possible since a suboptimal solution may lead to profit losses.
Some microgrid customers might primarily seek their own energy cost minimization and have diverse needs, although they all might benefit from the common objective of lowering operating costs of their feeder. In a residential microgrid, for example, one household might have at one particular moment increased electric energy needs, for example for cooking, while another household might need no electricity at all, because all its tenants are absent. Both households would like to sell the extra power produced locally to the grid, but it is unlikely that they would accept remote control of their production. In this case, the number of nodes might increase significantly and a neighborhood might have dozens of households or installed DGs. In such cases, it is not possible to have a dedicated communication system, but existing infrastructure should be used. Furthermore, the optimization problem is becoming extremely complex, as a result of specific characteristics. For example, it is extremely complicated to model the comfort requirements in each household or to include all the special technical constraints of all appliances in a single optimization problem. The decentralized approach suggests that this type of constraint and sub-problem should be solved locally in each household or DG.

2.5 Overview of the potential microgrid benefits

Depending on its operation strategy (see figure 2.3.1), a microgrid can provide a large variety of economic, technical, environmental, and social benefits.
The wholesale level is the price at which the microsource units sell the energy to the grid. On the other hand, the retail level is the price at which consumers buy the energy. The wholesale price is lower than the retail price. Economic benefits are mainly due to locality benefit attributed to the creation of an internal "over-the-grid" energy market within the microgrid, where microsource units could sell at prices higher than the prices at wholesale level and end consumers could buy at prices lower than the retail level. Local transactions keep profits within the community and encourage reinvestments in additional renewable generation [12]

A microgrid can also potentially improve the technical performance of the local distribution grid mainly in the following aspects:

- Energy loss reduction due to decreased line power flows (an estimated 5% of electricity created in the US is lost in transit [6]).
- Improved voltage quality via coordinated reactive power control and constrained active power dispatch.
- Relief of congested networks and devices, for example during peak loading through selective scheduling of microsource outputs.
- Enhancement of supply reliability via partial or complete islanding during loss of main grid.

The actual level of technical benefits depends mainly on two factors: the optimality of microsource allocation and the degree of coordination among different users.
Environmental benefits of a microgrid can be expected from two aspects: shift toward renewable or low-emission (e.g. natural gas) fuels and adoption of more energy-efficient energy supply solutions (e.g. combined heat and power applications).
Social benefits of microgrids can be mainly expected from:

- Raising public awareness and fostering incentives for energy saving and GHG emission reduction.
- Creation of new research and job opportunities.
- Electrification of remote or underdeveloped areas.

All of these listed impacts, however, can be seen as long-term effects.
The economic, technical and environmental benefits of the microgrids are represented in the figure 2.5.1. Each benefit item is mapped to the related stakeholder with dotted lines.

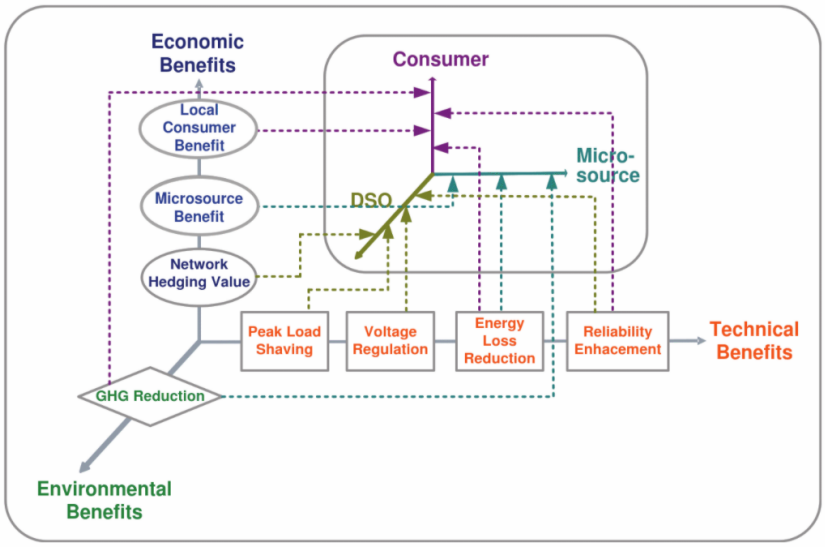


Figure 2.5.1: Overview of the economic, technical and environmental benefits of the microgrids [7]

2.6 Microgrid development

Finally, the roadmap for the microgrid development is displayed in the figure 2.6.1. It is true, that cost, policy and technology barriers have largely restrained the wide deployment of microgrids in distribution networks owing to their limited commercial appeal or social recognition. However, these three barriers are currently undergoing considerable changes – they are very likely to turn into key enablers in the future, eventually leading to a widespread microgrid adoption worldwide.

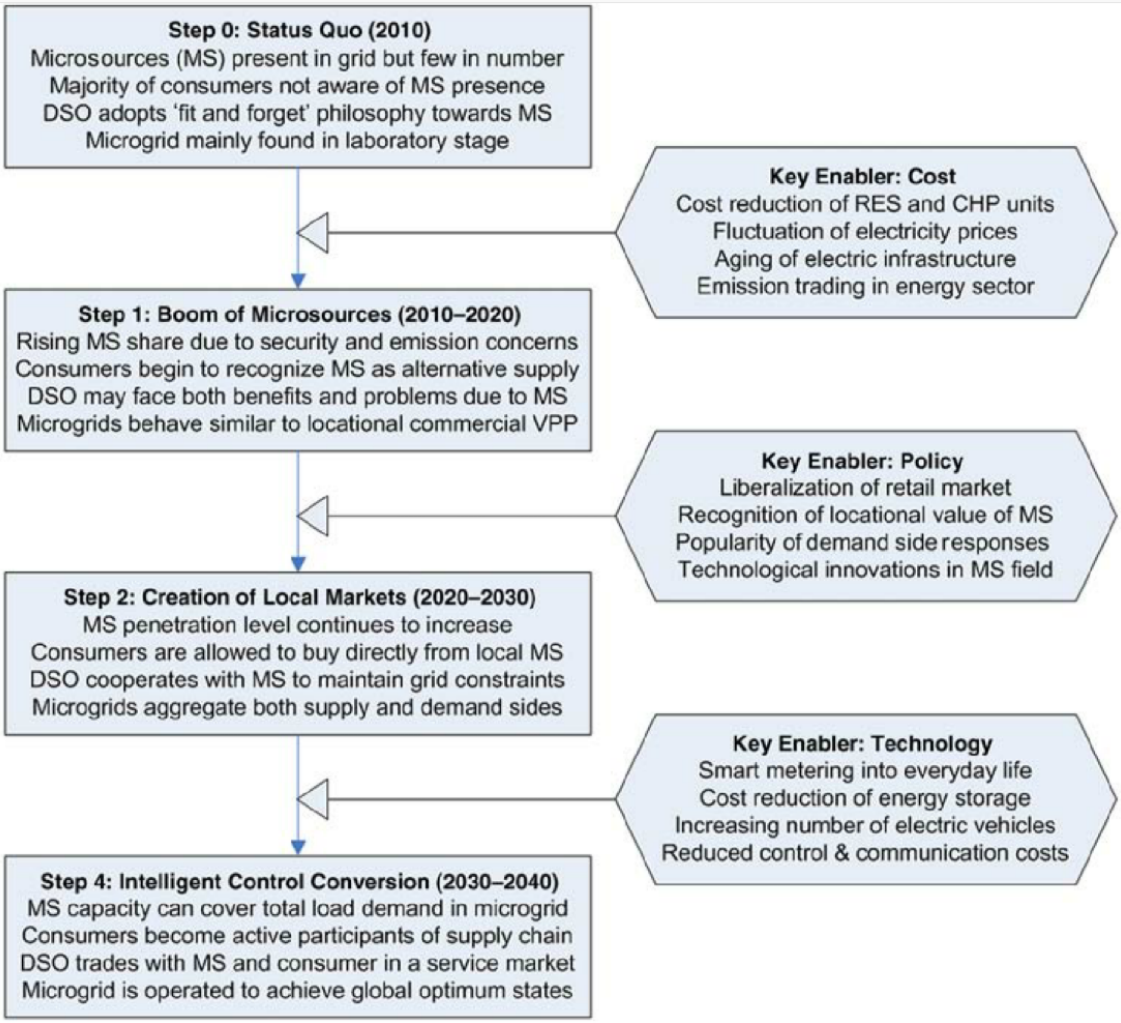


Figure 2.6.1: Roadmap for microgrid development [7]

Chapter 3

Blockchain in the energy sector

One of the objectives of this work is to find if WeSmart could use a blockchain as a trading platform. The goal of this chapter is to make a review of the blockchain in the energy sector and to see how and in which context blockchains have been implemented with microgrids.

The number of publications is one of the important indicators for evaluating the development status of a scientific field, while the annual change curve of publications can reflect the development process of the research topic, and then reveal the development stage of basic research. In 2020, Qiang Wang et al [34] made an analysis of the evolution of the blockchain technology in the energy sector. Here is the result of their work with a slight update of the number of the articles published in 2020 and 2021.

Figure 3.0.1 shows the annual number of articles of blockchain studies in the energy sector published in the Scopus database from 2014 to 2021

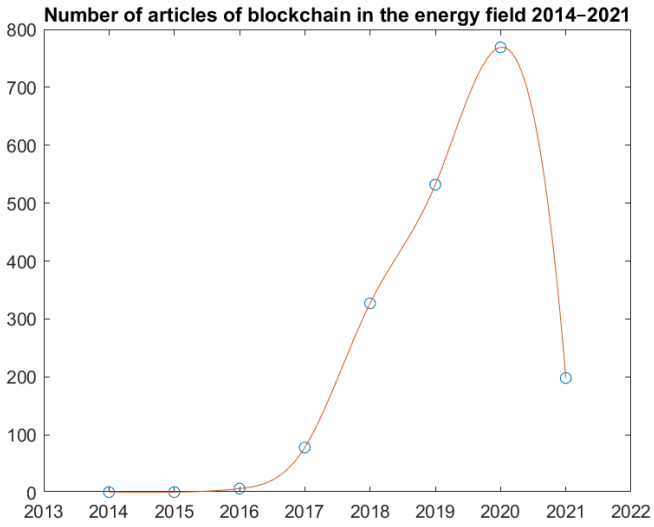


Figure 3.0.1: Number of articles of blockchain in the energy field 2014–2021

As can be seen from this figure, the Scopus database included the first article related to the energy blockchain in 2014. Those numbers illustrate the fact that although the blockchain technology appeared in 2008, it took some times before the use of it in fields different from the financial one. As of 2016, 7 articles were included in the Scopus database. It may be due to the fact that scholars have not fully realized the application value of blockchain in the energy field. This situation has changed from 2017, with a lot of blockchain researches on energy emerged. In 2017, 2018, 2019 and 2020, Scopus database included 78 articles, 327 articles, 532 articles and 769 articles on the blockchain in the energy field. More precisely, 2017 was a turning point. After this date, blockchain technology has been widely used in the energy field, and attracted more researchers' attention, resulting in an explosive growth in the number of literatures. In this sense, the related research of blockchain technology is no longer limited to the technical level, but is expanding to a wider field, and energy is one of the important ones. Note that the statistics in 2021 are not complete (198 articles on April 6, 2021), but according to the current growth rate of the literatures, blockchain research involving energy has great developmental potential in the coming days.

In 2019, Merlinda Andoni et al. [21] identified more than 140 blockchain innovation projects and research initiatives in the energy space. They found that approximately one in three use cases was about decentralized energy trading, which includes wholesale, retail and P2P energy trading initiatives. The second most popular category was cryptocurrencies, tokens and investment accounting for one

in five use cases. This was followed by "IoT, smart devices, automation and asset management" and "metering, billing and security", accounting for 11% and 9% of total use cases, respectively. Other projects make up 6–7% of the total (see figure 3.0.2).

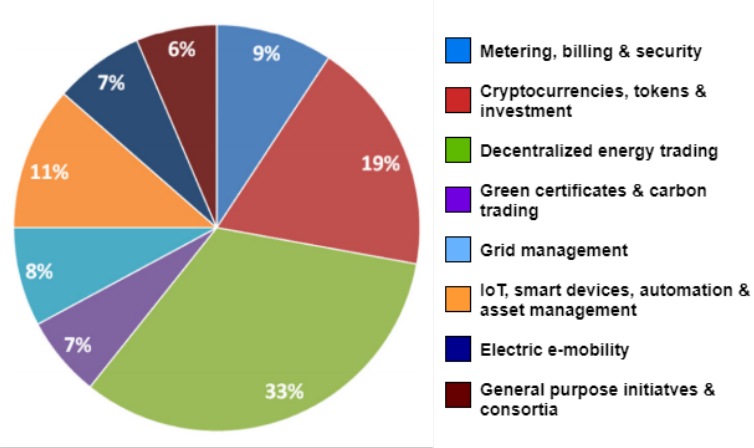
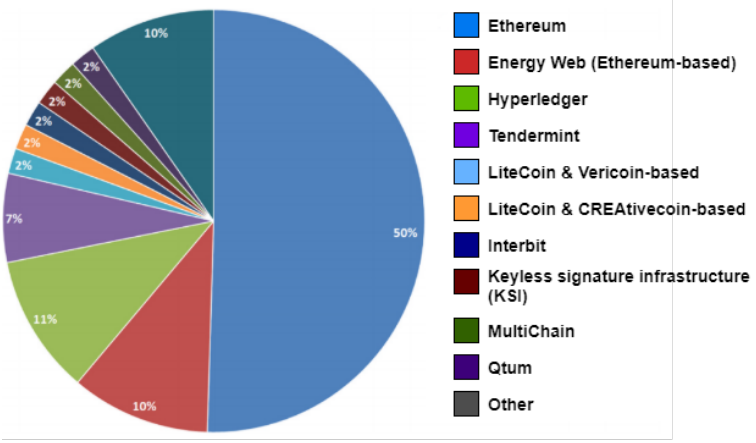
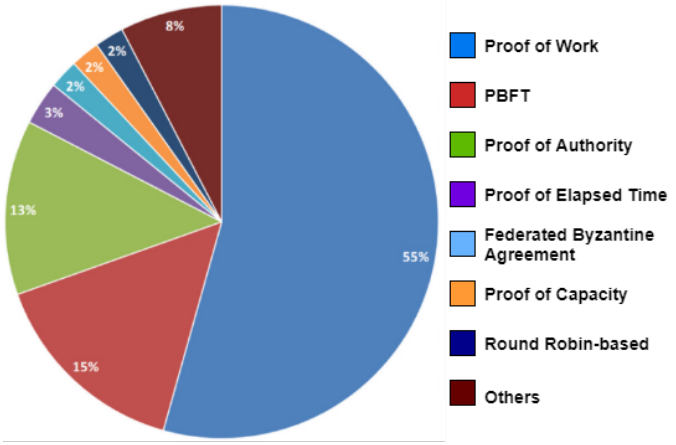


Figure 3.0.2: Blockchain use case classification according to their activity field: results derived from a study on 140 blockchain initiatives in the energy sector [21]

They also classified blockchain activities according to the platform and consensus algorithms used, wherever information had been made publicly available (see figure 3.0.3). 60% were as a starting point developing solutions based on Ethereum, while 55% had used PoW algorithms. Energy Web (also an Ethereum-based blockchain specially designed for the energy sector) was being explored by 10% of the projects which had publicly revealed information on the platform of their preference. Energy Web uses PoA consensus, a preferred solution for energy utility companies. Other popular platforms included Hyperledger and Tendermint. They finished by saying that future development projects might see a switch towards more faster and more energy-efficient blockchains that will be exploring PoS or BFT-type of solutions.



a) Blockchain use cases in the energy sector according to blockchain platform used: results derived from a study on 140 blockchain initiatives in the energy sector [21]



b) Blockchain use cases in the energy sector according to consensus algorithm used: results derived from a study on 140 blockchain initiatives in the energy sector [21]

Figure 3.0.3

3.1 Microgrids and blockchain

As explained by Huawei Mei et al [31], the microgrids and the blockchains shared some interesting characteristics. Moreover, the blockchain is able to ensure some essential features needed for the operation of the microgrid. Here are the major points of this study:

- Shared and not tamperable: the decentralized structure of blockchain fits into the decentralized approach for control and business processes in a microgrid. Moreover, thanks to its cryptographically-secure techniques, blockchains are considered immutable. As a result, participants can rely on a shared transparent and trustable database while the use of smart contracts permit to manage energy and financial flow among participants in a transparent and secure manner. The need of a central trusted entities that would maintain a common database and manage the energy and financial transaction is so avoided and the threat of a single point of failure is eliminated.

- Identification and privacy: thanks to asymmetric cryptography, each participant can use its private key as a digital signature to approve any new data that is recorded in the blockchain. Thanks to this mechanism, the members of the microgrids can verify the authenticity and the integrity of the data stored in the blockchain.

Market participants are allowed to poll their current balance encrypted by their public keys and perform decryption by using their private keys. Because there are the only one to have the private key, the privacy of this information is ensured.
Thanks to private blockchain, it is also possible to restrict the access to the blockchain to legitimate users.

- Payment: after energy has been shared, prosumers have to be paid. This is possible in the blockchain technology thanks to cryptocurrencies.

If the experience gained with blockchains in the financial sector are applied to the energy context, blockchain technology appears capable of enabling a decentralized energy supply system [18], [69], [66]. It may be possible to radically simplify today's multi-tiered system, in which power producers, transmission system operators, distribution system operators and suppliers transact on various levels, by directly linking producers with consumers. In this case, blockchain systems would initiate and transit transactions whilst recording them in a tamper-proof manner. All transactions made between individual parties would directly be executed through a peer-to-peer network. A fully decentralized energy transaction and supply system as illustrated in figure 3.1.1 can be considered to represent the ultimate level of energy-related blockchain applications from a theoretical perspective.

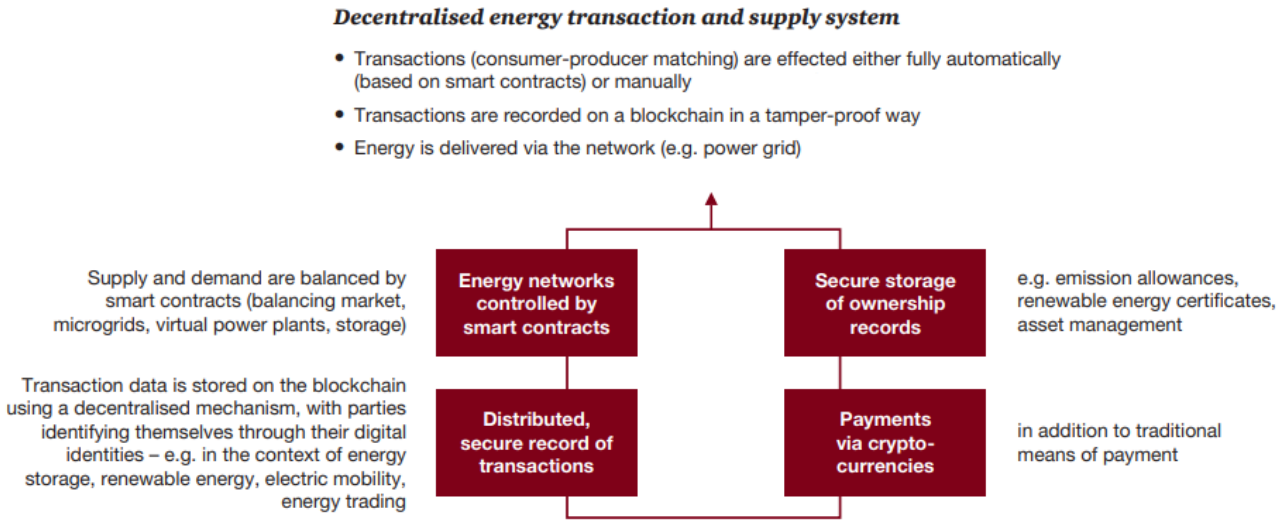


Figure 3.1.1: Cornerstones of a decentralised energy- transaction and supply system [69]

In the rest of this chapter, the Brooklyn microgrid project will be detailed as an example of existing peer to peer energy exchange mechanism based on the

blockchain concept. However, other projects are also developed such as Jouliette [63]: a blockchain supported showcase microgrid collaboration between Amsterdam's De Ceuvel sustainable office park, Dutch DSO Alliander, and an energy solutions developer called Spectral. Thanks to the use of the blockchain, customers could use cryptocurrencies to pay for the energy supplied. Projects such as NRGcoin [8] or Greeneum [53] plant to implement this feature.

3.2 The Brooklyn case [16]

In April 2016, the Brooklyn microgrid (BMG) run by LO3 Energy was the first study case project that combined the blockchain technology with the concept of microgrid. At this moment, severe weather events (e.g. hurricanes, heat waves, etc.) increased, which raised operation issues of the outdated electrical grids in Brooklyn. This project, implemented on top of the existing physical grid infrastructure, aims at increasing the local security of power supply by integrating renewable generation. In this project, prosumers can sell their energy surplus directly to their neighbours by use of Ethereum-based smart contracts and PBFT consensus, implemented by Tendermint. The first trial included 5 prosumers and 5 neighbouring consumers and resulted in the first ever energy transaction recorded in blockchains worldwide. The project consists of two main components:

1. The virtual community energy market platform: This platform provides the technical infrastructure for the local electricity market. It is based on a private blockchain using the Tendermint protocol.

2. The physical microgrid: An electrical microgrid is build in addition to the existing distribution grid. The physical microgrid acts as back-up to prevent power outages. By uncoupling from the traditional grid, it can operate in island mode. Then, critical facilities (e.g. hospitals) receive energy at fixed rates. Residences and businesses have to bid on the microgrid's remaining power.

The figure 3.2.1 represents the topology of the Brooklyn microgrid.

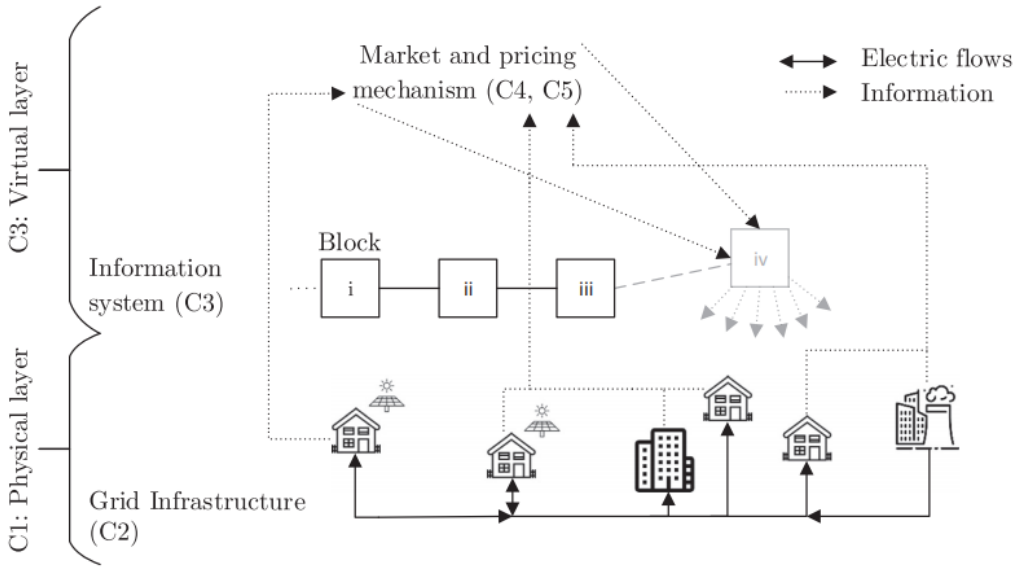


Figure 3.2.1: BMG topology [16]

As explained earlier, prosumers can sell their energy surplus directly to their neighbours. Microgrid users interact with the virtual platform by specifying their individual price preferences in the form of willingness to pay or sell electricity (see section 3.2.1). The platform can display location-specific and real-time energy prices. The ledger records contract terms, transacting parties, volumes of energy injected and consumed as measured by metering devices and crucially the chronological order of transactions. In addition, payments are automatically initiated by self-executed contracts. Every member of the community can have access to all historic transactions in the ledger and verify transactions for themselves.
More than 300 houses and small businesses, including around 50 PV prosumers and one small wind turbine generator, have signed for the next phase of development, which aims to achieve fully automated transactions. Microgrid members will be able in the future not only to decide from whom to buy/sell energy tokens based on their price preferences, but also on other criteria that reflect their environmental or social values. For instance, a consumer can specify the maximum price he is willing to spend on locally produced renewable energy but he can also declare other preferences such as percentage of energy they are willing to purchase from local renewable energy or the main grid. Users can even prioritise selling/buying energy from friends, family or a specific neighbour [49].

3.2.1 Energy trading

Here is how the energy trading is working. The consumers constantly bid their maximum price limit for their preferred energy sources (e.g. local renewable energy). Prosumers bid the minimum price limit that they request for selling their generation on the microgrid market. For example, consumer bids a maximum price X1 for quantity Y1 and prosumer P offers quantity Y2 for a minimum price of X2. Similar to a merit-order dispatch, the highest bidder is allocated first, then the lower bidders are allocated. The last allocated bid price represents the market clearing price for this time slot. Consumers that do not undercut the clearing price will be supplied by additional energy sources (i.e. hyper-local energy or traditional "brown" energy) allocated over a similar bidding mechanism or predefined prices.

3.2.2 Role of the blockchain

The blockchain implements the virtual layer of the BMG architecture (see figure 3.2.1). This virtual layer contents three components: the information system, the market mechanism and the pricing mechanism.
The information system is needed to connect all market participants and must provide equal access to the data for each market participant. In the BMG, secure smart meters write the energy generation and demand directly into the corresponding blockchain account. Buy and ask orders are then created according to this information and orders are sent to the market mechanism that can be sustained by smart contracts. The main objective of the market mechanism is to provide an efficient allocation of the traded energy by matching the market participants buy and sell orders. The market mechanism of the BMG is described in the section 3.2.1. In this double auction mechanism, the last allocated bid price represents the market clearing price which serves as pricing mechanism. Once the consumer and prosumer agreed on a price, payment is carried out and a new block (Block iv in the figure 3.2.1) is added to the blockchain. This block includes all current market informations.

3.3 Blockchain opportunities and risks in the energy sector

Thanks to its various characteristics, blockchain can bring a lot of opportunities in the energy sector. However, blockchain technology is still in its infancy at present, which means that it comes with a range of uncertainties and risks. This next table reviews the different opportunities and risks that the blockchain can bring in the

energy sector.

Opportunities	Risks
Lower transaction costs due to the cutting out of intermediaries	Currently high transaction costs for public blockchain systems
Transactions are generally made more simple (documentation, contracts, payment)	Possibly lack of acceptance on the part of consumers
Greater transparency thanks to decentralized data storage	No authority in the case of disputes
Flexible products (tariffs) and supplier switching	Lack of long-term experience
Strengthening of prosumers thanks to independence from central authority (direct purchases/sales of energy)	Technical problems with initial applications possible to start with

3.4 Legal concern

Several projects using a blockchain for energy trading have been developed. The Brooklyn project is an example of microgrid that uses the blockchain technology to record the energy transactions but also for the market mechanism. However, it is not possible to deploy this kind of project in Europe due to the General Data Protection Regulation (GDPR) [22]. Because the households' energy consumption data are considered as personal data as they could reveal much of personal habits, those data must follow the rules of the GDPR [30]. Two of them are of special interest in blockchain application : the "right of rectification" and the "right of erasure". The immutability aspect is one of the biggest advantage of the blockchain from a technological point of view but it is critical from the data protection perspective. In order to respect the GDPR, raw data can not be put on a blockchain. Although it has been demonstrated that the blockchain could be an efficient tool to develop the renewable energy communities, some legal points have to be developed in order to use it for energy trading. The next parts of this work then focuses on the use of the blockchain to guarantee the cloud integrity. As it will be explained, this application observes the GDPR because the blockchain is used to store the hash of the raw data.

Chapter 4

Blockchain to ensure cloud integrity

One of the objectives of this work is to ensure the integrity of the WeSmart's data stored in the cloud. The goal of this chapter is to make a review of the literature in this domain.

[25] The fundamental properties of cloud computing in data protection are : confidentiality, integrity and availability. Together there are called CIA properties. Data confidentiality is the property that ensures that a particular data must not be disclosed to an unauthorized user. This is a passive attack because it does not damage the data and it remains often undetected. [15] Data availability is the property that ensures that the data are available by an authorized user anywhere and anytime. This property can be compromised temporarily in the case of a denial of service attack. However it can be solved with replication methods. Data integrity is the property that ensures that data are not modified. This property should be handle more carefully than the others because it can both damage data and remain undetected. The rest of this work will focus on data integrity. There are mainly two types of data integrity verification mechanisms : the provable data possession (PDP) or the proof of retrievability. The proof of retrievability is used to restore the data from a loss of the server [10]. The provable data possession is used to quickly verify if the data stored on the cloud is intact. The basics of this method are proposed by Deswart et al [2]. Before to store data on the cloud, the user uses a Hash-based Message Authentification code (HMAC) to calculate the Message Authentification Code (MAC). This MAC can be seen as the fingerprint of the data. Then, the user saves this value locally and securely. To check the integrity of the data, the user downloads the data on the cloud, computes the MAC of these data and compares the result with the value stored locally. If they are the same, the integrity of the data retrieved from the cloud is confirmed. The

drawback of this technique is that directly downloading complete data may require a lot of resources.
In some cases where the client does not have the time, feasibility or resources to monitor their data, the PDP task can be delegated to a trusted third party authority [5]. Figure 4.0.1 shows such a framework. There are three objects : the client, the cloud storage server and a third party authority. The client stores its own data on the cloud and sends relevant information to the third party authority to verify the data integrity. The TPA can then perform some hashing algorithm on this information. When a data integrity verification is performed, the cloud sends proofs to the TPA. The TPA can then verify the integrity of the data on the basis of these proofs and the relevant information sent by the client. The weakness of these techniques is the third party authority. If the TPA colludes with the cloud, it can cheat the user. Moreover the TPA is vulnerable to accidental corruption or malicious forgery of provenance data. The next section presents articles that implement blockchains in order to ensure cloud data integrity while avoiding TPA.

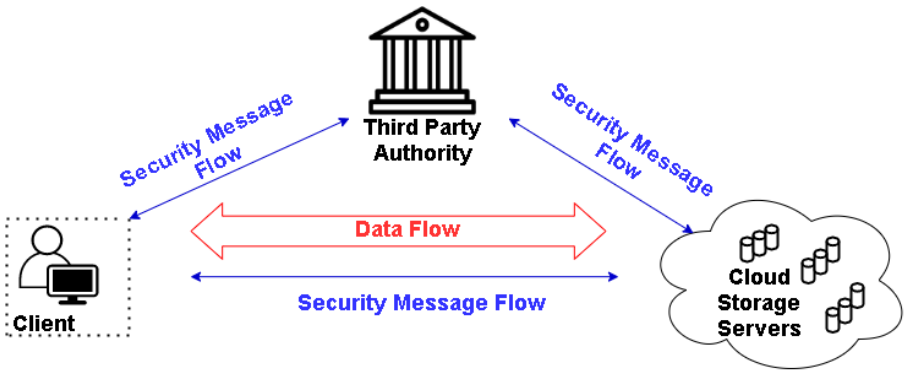


Figure 4.0.1: Data storage and integrity verification on the cloud [5]

4.1 Proof of data possession

This section reviews some articles of the literature that use a blockchain to perform a proof of data possession.

4.1.1 Hash of the entire data

Angelin E. et al. [25] proposed a model to rapidly check if the data has been tampered with in the cloud without the need of a third party. The figure 4.1.1 shows the architecture of the integration of blockchain into data storage in cloud. The cloud service provider (CSP) is responsible for the distribution of the user data among the virtual servers. D1, D2, ..., Dn represent the data id of the data

sent by the users and H1, H2, ..., Hn are their respective hash values generated by the server. H1', H2',..., Hn' are the new hash values that are calculated by the blockchain by linking with the previous blocks while adding every new block in the chain.
In this solution, the users send their data to the CSP. The CSP then send the data to any of its virtual servers. This server then generates the hash value of the given data and sends it back to the CSP. It will then send the client ID (user who has uploaded the data), the data ID and the hash value of the particular data to the blockchain.
This blockchain is shared with all the clients using that cloud. When adding a new block, the blockchain creates a new hash H' linked with the previous block. After every update, the blockchain is written in all the local copies of the clients.

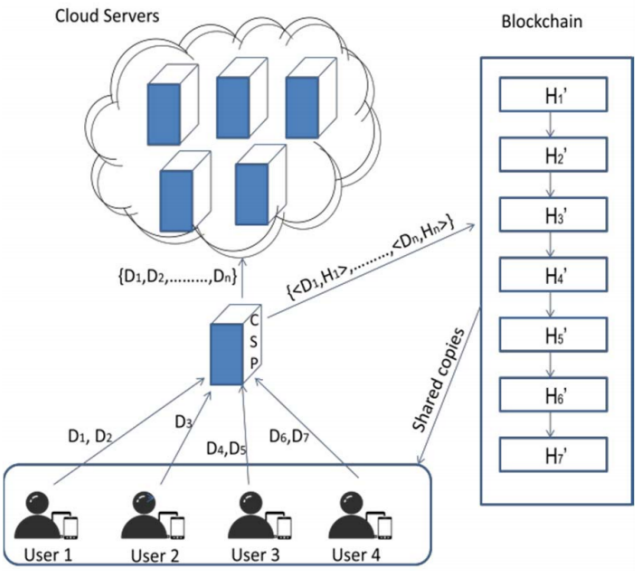


Figure 4.1.1: Role of the blockchain in the cloud computing [25]

When a user wants to make sure that its data has not been tampered with, the blockchain is invoked. The blockchain gets the data ID that is suspected to be corrupted and it insists the server to generate a hash value for the particular data. This new hash value is compared against the old hash value that is stored in the block. If these values don't match with each other, then the notification is sent to the particular client about corruption.
In the figure 4.1.2, an attacker tries to tamper the data D5. When the blockchain is invoked, the hash comparison is performed and the message about the corruption is sent to the respective client.

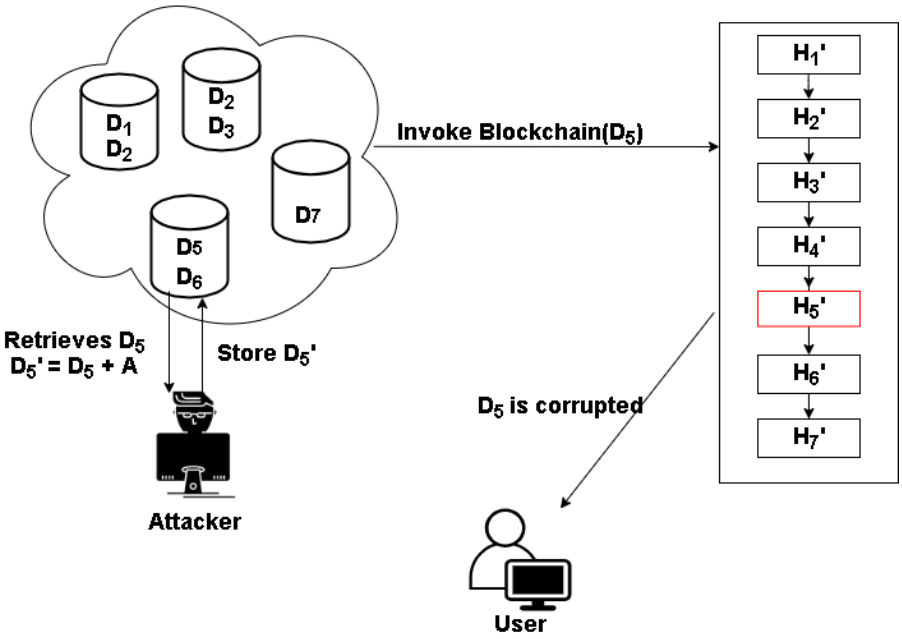


Figure 4.1.2: Data corruption [25]

4.1.2 Local computation of the hash data

While the hash of the data was made by the cloud storage service (CSS) in the previous section, Bin Liu et al. [15] distinguished two cases: the cloud storage service supports cryptographic has functionalities and the cloud storage service (CSS) does not support cryptographic hash functionalities.
In the first case, when the client needs to verify the data set integrity, the CSS computes the hash of the (entire) corresponding data and returns it to the client. After receiving the feedback from the CSS, the client queries the blockchain and gets the hash of the corresponding data. If the hashes obtained from the CSS and the blockchain are equivalent, then the data integrity is verified. On the other hand, if they are different, it means that the data of the cloud have been tampered with.
In the second case (when the CSS does not supports cryptographic hash functionalities), the principle is the same than for the first case. However, this time, the CSS returns the entire data and the client has to make the hash of those data. Again, if the hash computed by the user is equivalent to the hash stored in the blockchain, then the data integrity is verified.

4.1.3 Division of the data into shards

While in the previous section, the hash of the entire data is stored in the blockchain, Dongdong Y. et al. [20], proposed a method where the data are divided into shards. Thanks to the Merkle tree, data file can be verified by a small segment of the entire data shards, which is relatively small regardless of the size of the original file.
Here is the data integrity verification framework. There are three entities in this architecture: Clients, Cloud Storage Servers (CSS), and Blockchain. Clients upload their own data to the CSS and use the blockchain to verify data integrity. The overall workflow is divided into two stages. In the first step, the client slices its data into several shards, then uses these shards to construct a hash Merkle tree. In the second step, the client and CSS will agree on the hash Merkle tree. In the third step, the client will store the root of this hash tree denoted as *root1* on the blockchain. In the fourth step, the client uploads his data and public Merkle tree to CSS. In the fifth step, CSS return the address that stores the client's data to client.
There are also five steps in the verification phase. In the first step, the client will send a request to the CSS to verify a particular shard i. In the second step, CSS use hash function to calculate the hash i' of the shard i. In the third step, CSS send i' and the corresponding auxiliary information to the blockchain. In the fourth step, the smart contract on the blockchain calculates a new hash root denoted as root2, and compares root1 with root2. If they are equal, the data integrity has been guaranteed; otherwise, the data integrity has been corrupted. In the last step, the BC will return the verified result to the client.
In this framework, clients will place the root of the Merkle trees on the blockchain before uploading the data. Due to the non tamperability property in blockchain, any client or CSS can not modify the root stored on the blockchain, which makes integrity verification more credible.

Due to limitation of real-time requirement, this article proposes to verify only part of the shards in order to confirm the data integrity.
Here are the results of the impact of the total number of validated shards (also called sample size) on the cost and precision of verification. For the verification cost, a simple linear function can be used to express the relationship between the sample size N and the verification cost C:

$$C = c_0 + c_1 N$$

Where $c_0 > 0$ and $c_1 > 0$. c_0 represents the basic cost and c_1 represents the influence degree of sample size.
For the verification precision, let's suppose that the total number of data shards is

n, where there are f invalid (lost or tampered) shards, and the sample size is N. The variable V is used to represent the number of invalid shards detected in the sampled data, then the probability P_V represents at least one invalid shard has been detected, which is:

$$\begin{aligned} P_V &= P\{V >= 1\} = 1 - P\{V = 0\} \\ &= 1 - (\frac{n-f}{n})^N \end{aligned}$$

The ideal situation is to spend as little verification cost as possible and obtain as high verification precision as possible. However, the verification cost and the verification precision are in contradictory relationship. An optimal sample size to tradeoff the contradiction between the verification cost and the verification precision has to be found. Therefore, a Loss function $L(N)$ is proposed to comprehensively consider the impact of sample size on verification cost and verification precision:

$$L(N) = C + \lambda \frac{1}{P_V} \tag{4.1.1}$$

$$= c_0 + c_1 N + \lambda(\frac{1}{1 - (\frac{n-f}{n})^N}) \tag{4.1.2}$$

where $N \in]0, n], c_1 > 0, c_0 > 0$. The relationship among loss, cost and precision reflected by the loss function should conform to the actual situation. That is, the higher the cost, the greater the loss, i.e. the loss is proportional to the cost. The higher the precision, the smaller the loss, i.e. loss and precision are inversely proportional. The equation 4.1.2 is adopted to express the relationship among loss, cost and precision in a simplified way. c_0, c_1, n, f being parameters, the goal is to find an optimal N in order to make $L(N)$ minimum. In this equation, λ balances the importance between verification cost and verification precision. In the paper, $\lambda = 1$ and $c_0 = 0$ has been set. Two sets of experiments have then been set. In the first set of experiments, $n = 10000$, $c_1 = 0.053$ and four different values of f have been set : $\frac{f}{n} = 0.001$, $\frac{f}{n} = 0.002$, $\frac{f}{n} = 0.01$, $\frac{f}{n} = 0.05$.

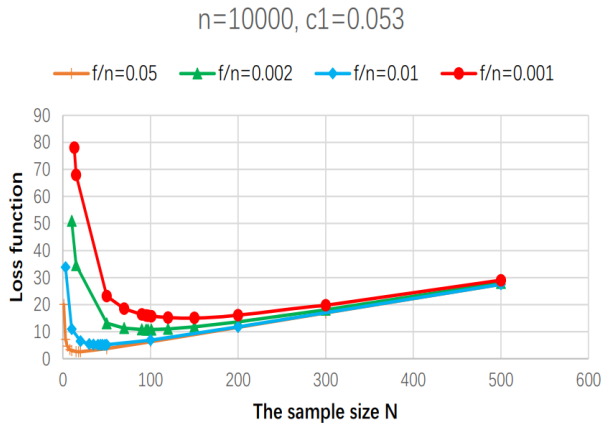


Figure 4.1.3: Relationship between Loss function $L(N)$ and the sample size N when $\frac{f}{n}$ changes [20]

In the second set of experiment, $n = 10000$, $\frac{f}{n} = 0.002$ and four different values of c_1 has been set : $c_1 = 0.7$, $c_1 = 0.4$, $c_1 = 0.1$, $c_1 = 0.01$.

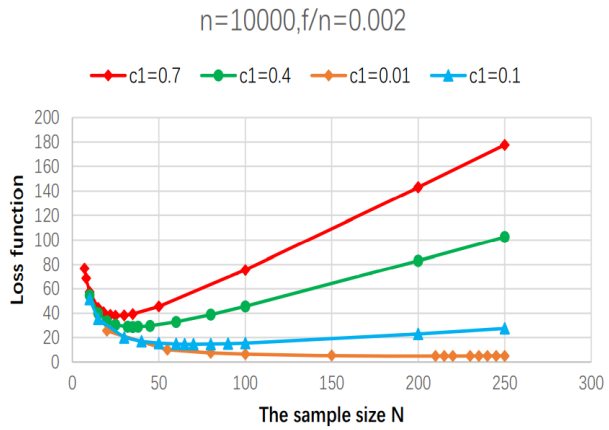


Figure 4.1.4: Relationship between Loss function $L(N)$ and the sample size N when c_1 changes [20]

The figures 4.1.3 and 4.1.4 both show that as the sample size N increases, the value of the Loss function $L(N)$ first decreases, reaches a minimum value and then starts to increase continuously. Thus, there exists a most appropriate value of N leading to the minimum value of $L(N)$. From figure 4.1.3, it can be observed that the more shards fail, the smaller the optimal sample size. It can also be seen that as $\frac{f}{n}$ increases, the minimum value of the Loss function $L(N)$ decreases so the overall validation performance of this system increases as the number of failed shards increases. From figure 4.1.4, it can be detected that when c_1 increases, the optimal

sample size decreases while the minimum value of the Loss function increases. It means that when the weight of verification overhead increases, the optimal sample size decreases, while the overall validation performance of this system decreases.

4.1.4 ProvChain to ensure the integrity of the provenance data

Liang et al [14] designed and implemented ProvChain, an architecture to collect and verify cloud data provenance, by embedding the hash of the provenance data into blockchain transactions. ProvChain records the operation history as provenance data which will be hashed into Merkle tree node. A list of hashes of provenance data will constitute a Merkle tree and the tree root node will be anchored to a blockchain transaction. A list of blockchain transactions will be used to form a block and the block needs to be confirmed by a set of nodes in order to be included in the blockchain. ProvChain uses the Multichain blockchain that provides an open-source permissioned blockchain network.

ProvChain achieves the following objectives :

- **Real-time cloud data provenance :** User operations are monitored in real time to collect provenance data, which will further support access control policy enforcement and intrusion detection.
- **Tamper-proof environment :** Data provenance record is collected and its hash is published to the blockchain network which protects the provenance data. All data on the blockchain are shared among the nodes. ProvChain builds a public time-stamped log of all user operations on cloud data without the presence of a trusted third party. Every provenance entry is assigned a blockchain receipt for future validation.
- **Provenance data validation :** The hash of the data provenance record is published globally on blockchain network, where a number of blockchain nodes provide confirmation for every block. ProvChain uses blockchain receipt to validate every provenance data entry.

To achieve the above objectives, the authors adopted the below methods to design ProvChain's architecture:

- Monitor user activities in real time using hooks and listeners so that every user operation on files will be collected and recorded for generating provenance data.
- Store all hashed data operations in form of blocks in the blockchain. Every node on the blockchain can verify the operations by mining the block so that data provenance is authentic and tamper-proof.

- Provenance auditor validates provenance data by retrieving transactions from the blockchain network by using blockchain receipt which contains block and transaction information.

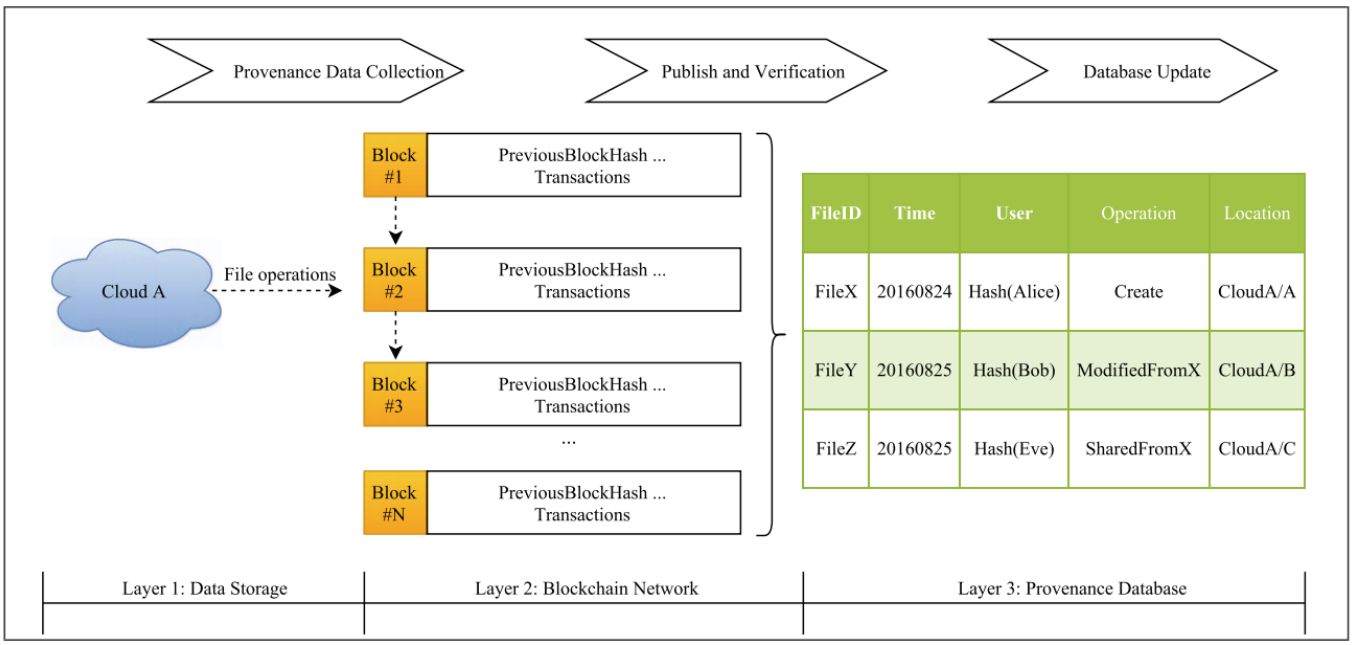


Figure 4.1.5: ProvChain architecture [14]

The figure 4.1.5 represents the storage architecture of the solution.

The global working of ProvChain is illustrated in the figure 4.1.6. The user first registers to the Cloud Server Provider (CSP) (1). After that, the user performs actions on the data files stored in the cloud (2) and the following elements get recorded: the date, the hour and the file that has been modified but also the ID of the modifier and the type of modification. Together, those informations form the provenance data. The modifications are then done on the cloud and the hash of the provenance data is sent and stored in the blockchain (3). The provenance data are also updated locally in the provenance database (4).
When the provenance data has to be validated (5), the hash of the provenance data contained in the provenance database is compared with the hash of the blockchain (6). If there are similar, then the provenance data are validated (7).
In this architecture, the role of the blockchain is to ensure the integrity of the provenance data.

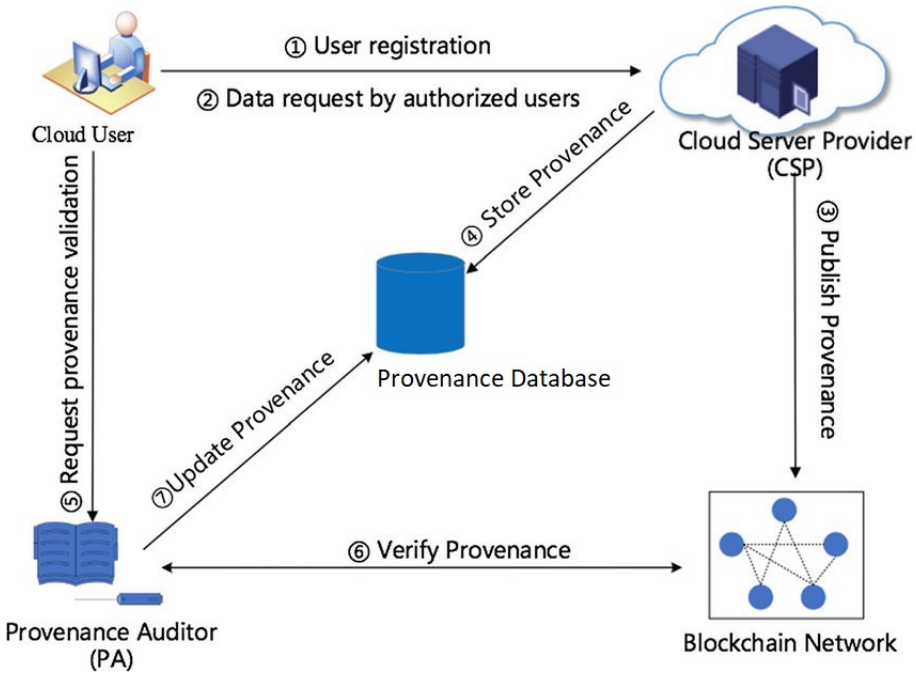


Figure 4.1.6: ProvChain system interactions [24]

4.2 Proof of retrievability

Ari Juels et al. [3] described a POR protocol that encrypts a file F and randomly embeds a set of randomly-valued check blocks called sentinels. The use of encryption here renders the sentinels indistinguishable from other file blocks. The verifier challenges the prover by specifying the positions of a collection of sentinels and asking the prover to return the associated sentinel values. If the prover has modified or deleted a substantial portion of F, then, with high probability, it will also have suppressed a number of sentinels. It is therefore unlikely to respond correctly to the verifier. To protect against corruption by the prover of a small portion of F, they employ error-correcting codes. $\tilde{F}$ refers to the full, encoded file stored with the prover.

Here are two scenarios that illustrate the intuition behind the POR protocol:

Example 1. Suppose that the prover, on receiving an encoded file $\tilde{F}$, corrupts three randomly selected bits, β_1, β_2, β_3. These bits are unlikely to reside in sentinels, which constitute a small fraction of $\tilde{F}$. Thus, the verifier will probably not detect the corruption through POR execution. Thanks to the error-correction present in $\tilde{F}$, however, the verifier can recover the original file F completely intact.

Example 2. Suppose conversely that the prover corrupts many blocks in $\tilde{F}$ (for example 20% of the file). In this case (absent very heavy error-coding), the verifier is unlikely to be able to recover the original file F. On the other hand, every sentinel that the verifier requests in a POR will detect the corruption with probability about 1/5. By requesting hundreds of sentinels, the verifier can detect the corruption with overwhelming probability.

The use of an error correcting code (ECC) is a common method to retrieve an original file from being slightly modified. The central idea behind this concept is that the sender encodes the message with redundant information in the form of an ECC. The redundancy allows to detect and correct a limited number of errors that may occur anywhere in the message. This technique is used a lot in the telecommunication sector for controlling errors in data when they are sent over an unreliable or noisy communication channel [52].
A simplistic example of ECC is to repeat each data bit with a factor of 3. This is known as a (3,1) repetition code and it is illustrated in the tabular below.

Triplet received	**Interpreted as**
000	0 (error-free)
001	0
010	0
100	0
111	1 (error-free)
110	1
101	1
011	1

Part II

Chapter 5

Proposed implementation

This part explains the solution intended to the WeSmart company. The current operation and the needs of the company have been analysed thanks to the participation of Francois Bordes, CEO of WeSmart and Thibaut Piraux, product owner in WeSmart.

WeSmart currently stores their consumer related data in a cloud. A weakness of this implementation is that the data integrity can not be insured. In this context, WeSmart asked us to provide a way to verify that their data stored in the cloud are not corrupted. The company also wanted the solution to be independent of the current implementation. Our solution has to be seen as an independent security layer that is added on top of the existing application.

To answer this demand, we introduced the concept of blockchain in the current implementation of WeSmart and we used the principle of the chapter 4 which is to store the hash of the raw data in the blockchain.
The next section describes our solution in more details. After that, the comparison of the existing tools to create a blockchain is achieved. Finally, the tool that we chose is described in more details.

5.1 Overview of our solution

There are currently three kinds of members in the WeSmart architecture: consumer, producer and prosumer. When a producer or a prosumer produces more than it consumes, the extra energy is distributed to the other members of the energy community. This distribution is defined by a repartition key determined at the creation of the energy community. This situation is represented in the figure 5.1.1. Data flow are represented by the links with a "i" symbol and energy

flow by the links with a lightning. In this architecture, members of the energy community exchange energy and the production/consummation information is sent to WeSmart. WeSmart then processes this information and stores them on the cloud. To insure that data can not be attacked during the transmissions between the users and the WeSmart node or between the WeSmart node and the cloud, we advise to use the TLS protocol to secure the transmissions.

The goal of this work is to provide an additional security layer on top of the existing operation. More precisely, the goal of this work is to find a way to detect the data corruption that can happen in the cloud.

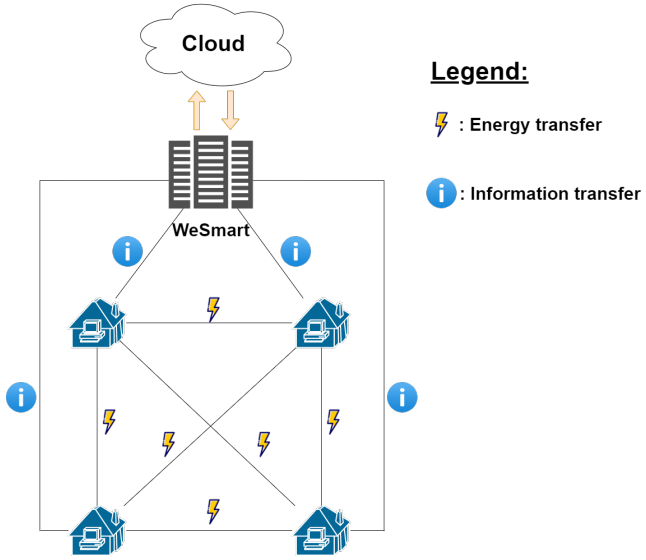


Figure 5.1.1: Overall architecture of the renewable energy community

As explained in the section 3.1, the decentralized structure of blockchain fits into the decentralized approach for control and business processes in the renewable energy community. For that reason, the concept of the blockchain can easily be deployed in the current solution. As represented in the figure 5.1.2, the network is fully peered [1] (each participant is connected to the other) and each participant stores the entire blockchain. For this reason and because the blocks contained in the blockchain are linked one with the others, a change in the content of one block will be detected by the other participants (also called nodes). In the proposed solution, nodes of the blockchain store the hash of the data. Indeed, every participant of the blockchain has access to all the information contained inside. It is then not

[1]Note that if a node is behind a NAT, its NAT table should be modified to make this node reachable from the outside of the network. This is explained in more details in the section 11.14 of the annexes.

acceptable for the data confidentiality to put raw data in the blockchain. An other reason is that the GDPR stipulates that personal data can not be directly inserted on the blockchain due to its immutable character (see section 3.4).

The security of the links between blockchain nodes is part of the solution and is realized thanks to a TLS like mechanism (cf. section 6.2.4)

The proposed solution works jointly with the existing solution : the raw data are stored on the cloud and a cryptographic hash of the raw data are stored on the blockchain.

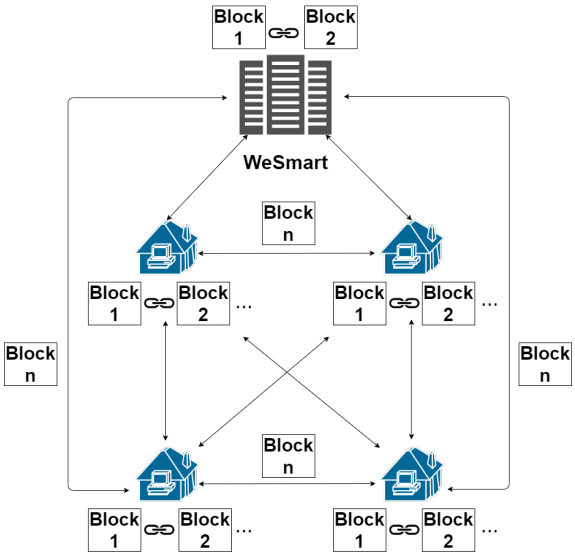


Figure 5.1.2: Overall architecture the security layer

The figure 5.1.3 represents the message flow of our solution. First, WeSmart sends their raw data (I_1) to the cloud and the hash of those data ($H(I_1)$) to the blockchain. I_1 typically contains a list of the energy consumed by each member of the energy community during the last 15 minutes. After a while, when the company wants to verify the integrity of the data, it sends a request to the cloud. The cloud then answers with the corresponding data that is possibly modified (I'_1). WeSmart also queries the blockchain for the hash of the wanted data. The blockchain then replies with $H(I_1)$. Because the ledger of the blockchain is immutable (characteristic of the blockchain), the value returned by the blockchain is the same than the value that WeSmart originally put on it. With I'_1 and $H(I_1)$ in hands, WeSmart compares the value of $H(I'_1)$ and $H(I_1)$. If they are the same, it means that $I'_1 = I_1$ and the integrity of the value put on the cloud is verified.

In this solution, all the communications "WeSmart-Cloud" and "WeSmart-blockchain" must be encrypted thanks to TLS. This solution also makes the hypothesis that WeSmart protects its cryptographic hash function. If an attacker

succeeds to modify this hash function, the integrity of the data is not guaranteed. Indeed if the attacker modifies some data in the cloud and also modifies the hash function to return the hash of the old value, then the comparison with the hash value in the blockchain remains true even if the data in the cloud are altered. To avoid this problem, one solution that WeSmart could choose is to perform the hash computation in a secure environment. An example of such environment is the SGX enclave used for the PoET (cf. section 1.6.5) that is used by developers to protect sensitive data or code from outside interference or inspection.

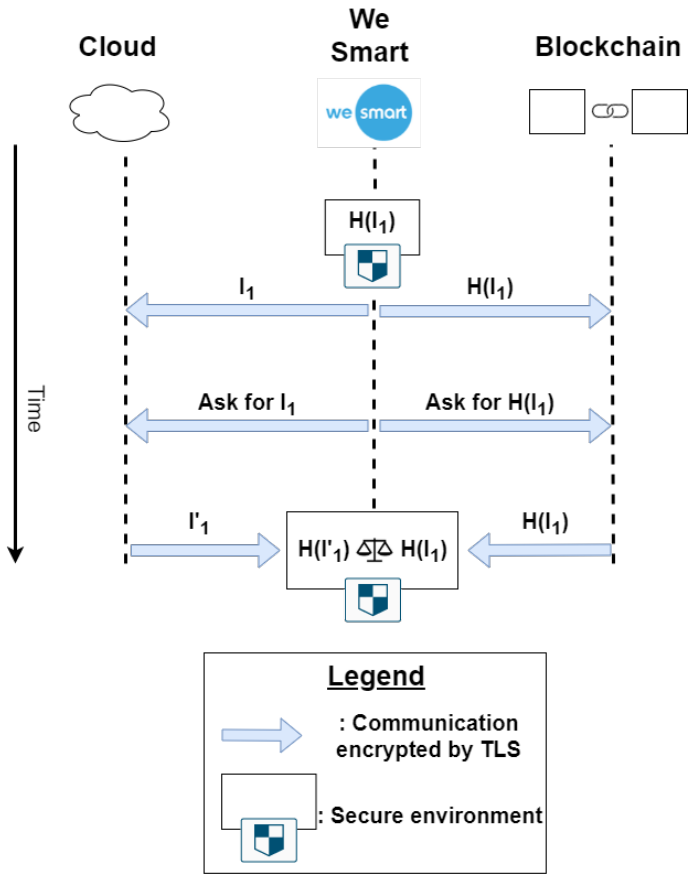


Figure 5.1.3: Time flow of the solution

In this context, the blockchain is used for its distributed and immutable characteristics. The immutable aspect ensures the correctness of the hashes returned by the blockchain. The distributed characteristic of the blockchain ensures that those hashes are still recoverable even if part of the nodes fails.

5.2 Analysis of the different tools available to implement a blockchain

Now that the general solution has been explained, the tool to implement the blockchain has to be chosen. This section performs a comparison of the different tools that are used in the blockchain domain. The first part of the section presents a table that synthesises those tools. Based on this table, the most appropriate tool is chosen.

Comparison of the different tools

	Language	**Smart contract**	**Consensus**	**Private Blockchain**
Ethereum	Solidity, Serpent, Lisp-LikeLanguage	Solidity, Vyper	PoW	No
MultiChain[42]	Python, C++,Go JavaScript , Ruby	No	Close to a PBFT	Yes
OpenChain	Javascript	JVM language *	Partionned Consensus	Yes
Corda	Kotlin	Java, Kotlin	Single Notary, Raft, BFT-SMaRt	Yes
Hyperledger Fabric	Java, Go, Node.js	Go, JavaScript Java, Solidity	Kafka, Raft	Yes
Hyperledger Sawtooth	Java, Python,Go, JavaScript, C++, Rust	Rust, JavaScript, Go, Python	PoET, PBFT,	Yes
Quorum		Solidity, Vyper	Raft, IBFT	Yes
EOS	C++, Java Python	C++	DPoS	Yes
Tendermint	Java, C++, Python, Go	Javascript, Rust, Soidity	Mixed PBFT-PoS	Yes

* : all Java Virtual Machine such as Java, Clojure, Groovy, Kotlin, Scala etc.

- **Ethereum :** This platform can not be used because it implements only public blockchain.
- **MultiChain :** In MultiChain, some of the features are not for free.
- **OpenChain :** Every Openchain instance only has one authority validating transactions. This authority acts as a third party trusted by all participants (that something that we want to avoid). Moreover, Openchain seems not very active at the moment.

- **Corda [40] :** Raft algorithm tolerates n/2 crashed nodes but it does not tolerate Byzantine nodes. The BFT-SMaRt is a high-performance Byzantine Fault Tolerant State Machine Replication. This algorithm executes correctly if less than 1/3 of the node are faulty. Corda is a platform build for finance but it can be adapted for other applications.

- **Hyperledger Fabric [41]:** Raft is the mostly used consensus algorithm in Hyperledger Fabric. This algorithm tolerates up to n/2 of crashed nodes but it does not tolerate Byzantine nodes. In Hyperledger Sawtooth, Kafka can still be used as a consensus algorithm but it becomes deprecated in the newest version of Hyperledger Fabric.

- **Hyperledger Sawtooth [78], [54] :** The main difference with Hyperledger Fabric is the used consensus. Hyperledger Sawtooth uses a PBFT algorithm that tolerates n/3 of Byzantine nodes. The drawback of this algorithm is that it is not scalable. However in our case the number of nodes in the blockchain quite low because it is the number of participant in the energy community. The non scalability of the PBFT is so not a big issue.

- **EOS :** The consensus used is a Delegated Proof of Stake.

- **Tendermint [36], [76], [88], [86], [38] :** While Tendermint implements an interesting consensus that seems to mix the best of the PBFT and of the PoS consensus, its biggest drawback is that it only implements the consensus and the network layer. An other tool is needed to use the application layer : the Cosmos SDK. Moreover, to use smart contract this application layer has to be extended with other tools. Tendermint is a flexible tool but the layering and multiplicity of the different tools makes its implementation more complex.

  The figure 5.2.1 is a representation of the architecture implemented in Tendermint. Note that the application layer communicates with Tendermint Core with a tool called ABCI.

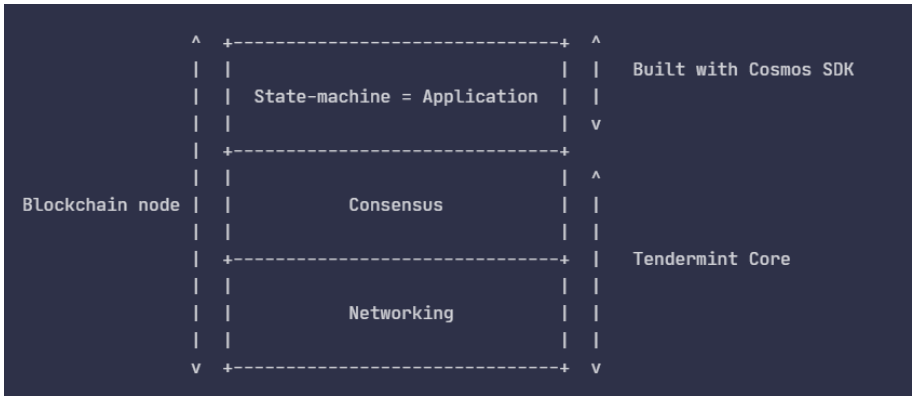


Figure 5.2.1: Overview of the architecture used in Tendermint [38]

5.3 Conclusion

For this project, a private blockchain is needed. Indeed, we only want participants of the renewable energy community to be part of the blockchain network. Solution such as Ethereum is so prohibited. In order for the blockchain to have the highest security level and because of the relative small size of the network, the BFT consensus is the more suitable for our solution. Indeed, as explained in the section 1.6.6, the advantage of this algorithm is that it can support up to $\frac{n-1}{3}$ Byzantine faulty nodes, with n, the total number of nodes in the network. Moreover, the transactions can be agreed upon and finalized without needing multiple confirmations. It means that the content of a block can be trusted even if it is the last one (remember that this is not the case for the Bitcoin where at least 6 more other blocks are needed to be produced in order to consider that the content of a block is valid). However, this algorithm suffers from two negative points: it is less scalable and it is prone to Sybil attack. Because of the relative small size of the current network of energy community, the scalability is not a real issue. The possible Sybil attack is also not relevant because in our case, the network is private and each node identity is known by WeSmart; each node represents a singular participant of the renewable energy community and it is unlikely that a node in the network operates multiple identities actively at the same time. Because of the high security level of the PBFT consensus and because its drawbacks are not relevant in the context of this work, this consensus seems to be the most well suited for the solution.
Since OpenChain seems not very active at this time, we think that is probably not the best choice to implement our solution. Tendermint is an interesting tool but it only implements the network and consensus layer. It exists other tools like Quorum or Corda that are more specific for building private blockchains. However, those

tools [43] are mostly used in the finance domain. On the other hand, Hyperledger seems one of the most popular tool for the creation of private blockchains. The consensus used for Hyperledger Fabric (Kafka and Raft) are crash fault tolerant. However, the two consensus used by Hyperledger Sawtooth (PBFT and PoET) are both Byzantine fault tolerant. For that reason, Hyperledger Sawtooth has been chosen for the implementation of the blockchain of this project. An other reason that brings us to use Hyperledger Sawtooth is that all the Hyperledger projects are open source. One of the big advantage of the open source projects is their high reliability. Indeed, since there are more eyes on them, they turns to be extremely robust. An other advantage for the possible future development and usage of this blockchain is that smart contracts can be implemented with this tool. Finally, as explained in the next chapter, Hyperledger Sawtooth is widely used in the industry sector. This confirms the relevance of this tool.

Chapter 6

Hyperledger Sawtooth

This chapter performs an explanation of the tool used for the implementation of our private blockchain.

6.1 Introduction

Hyperledger Sawtooth is an enterprise blockchain platform for building blockchain applications and networks. It is an open source project which results from the cooperation of Hyperledger and Intel in 2016. Hyperledger is a multi-project open source collaborative effort hosted by the Linux Foundation, created to advance cross-industry blockchain technologies.
Hyperledger Sawtooth is a widely used tool in the industry. Here are three examples of companies that use Hyperledger Sawtooth :

- **BondEvalue**[57], a Singapore-based fintech company that focuses on Asian bond markets, used Hyperledger Sawtooth to launch the world's first blockchain-based bond exchange.
- **Blockchain Technology Partners (BTP)**[95], a leading enterprise blockchain company, combined Hyperledger Sawtooth with Kubernetes to create Sextant. Sextant is a blockchain management platform that radically simplifies the deployment and ongoing management of blockchain-based enterprise applications.
- **ScanTrust**[58], a Software as a Service platform, connects millions of products to the internet by giving each of these products a unique digital identity. This company uses Hyperledger Sawtooth to offer supply chain traceability. It helps their clients, for example Cambio Coffee, to bring more transparency to their ethical trade business.

6.1.1 Distributed ledger

The characteristics of the Sawtooth blockchain are :

- **Distributed** : there is no central administrator. The database is shared among all the nodes and they all have the same information.
- **Immutable** : the blockchain database contains the history of all the transactions into blocks. These blocks are linked with hashes which makes an alteration of the blockchain difficult.
- **Secure** : all changes in the blockchain are performed by transactions that are signed by known identities.

6.1.2 Separation application - core system

The development of a project with Sawtooth is made simple thanks to the separation between the core system and the application domain. The knowledge of the underlying design of the core system is not required to develop an application. In fact, the implementation of an application is realised by writing a specific transaction family. This transaction family defines a data model for the application and a set of operations/transactions that are allowed on the blockchain. A high flexibility is given to the developer in the choice of the programming language. It it possible to implement the transaction family in Python, Java, C++, Go or Rust.

6.1.3 Modularity and permissioning

Sawtooth core system makes the application highly modular. It allows the application to choose the transaction rules, permissioning and consensus algorithms to fit its needs. Thanks to those possibilities, it is possible to deploy a private network or to deploy nodes with different permissioning. The information concerning the permissions and roles is stored in the blockchain. In this way, all the participants in the network have access to this information and there is no risk of modification of these data.

6.2 Architecture

6.2.1 Architecture overview

The figure 6.2.1 shows the architecture of a Sawtooth network. The different components are :

- The **Rest API** : this component makes the link between the client and the validator. It allows the client to interact with the validator using common HTTP/JSON standards. It provides a serie of functions (POST, GET) for submitting transactions and reading blocks.
- The **validator** : this component is responsible for validating batches of transactions, combining them into blocks and maintaining the consensus. The validator can communicate directly with the client or through the Rest API. It can communicate with the transaction processors and it is also linked with the validators of the other nodes to form the Sawtooth network. All the nodes communicate together thanks to their validator.
- The **transaction processors** : for each transaction family, there is an associated transaction processor. Its role is to validate and update the state based on the rules defined by the transaction family.

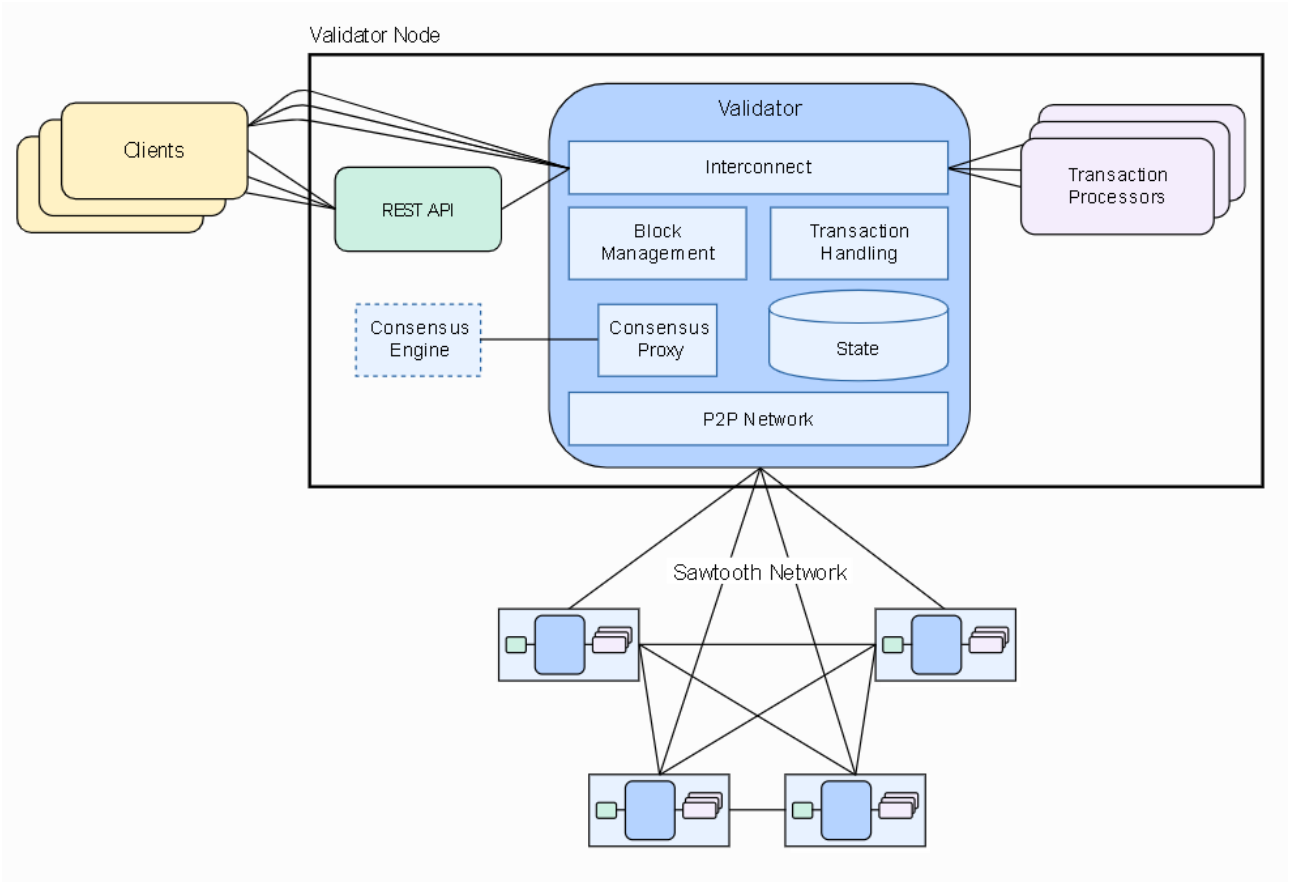


Figure 6.2.1: Architecture of a Sawtooth network

6.2.2 Data representation

The goal of the blockchain is to share a common global state between all the nodes. The consistent copy of data among all the node is ensured by the chosen consensus algorithm.

For flexibility, the global state is divided into different namespaces that are defined in the transaction family. This state is updated when the validator receives transactions. The block validation on each validator ensures that this transaction leads to the same state transition on all the nodes.

Addressable Merkle Radix Tree

States for all the transaction families are represented in the form of a single addressable Merkle Radix tree. The concept of Radix tree is explained in the annexes (cf. section 10.1).

Data are stored on the leaf nodes of the tree. Then, following the principle of a Merkle tree, the successive node hashes are computed from leaf to root. The root of the tree is so a hashed representation of the entire state. This state root hash is placed in the block header to perform the consensus on the expected version of the state. Indeed, if any leaf node is different between two states, the state root hash of the two states are different and the block is not considered valid.

The figure 6.2.2 shows the Merkle tree associated to a state. The tree has a maximum depth of 35 nodes and each node has up to $2^8 = 256$ child nodes.

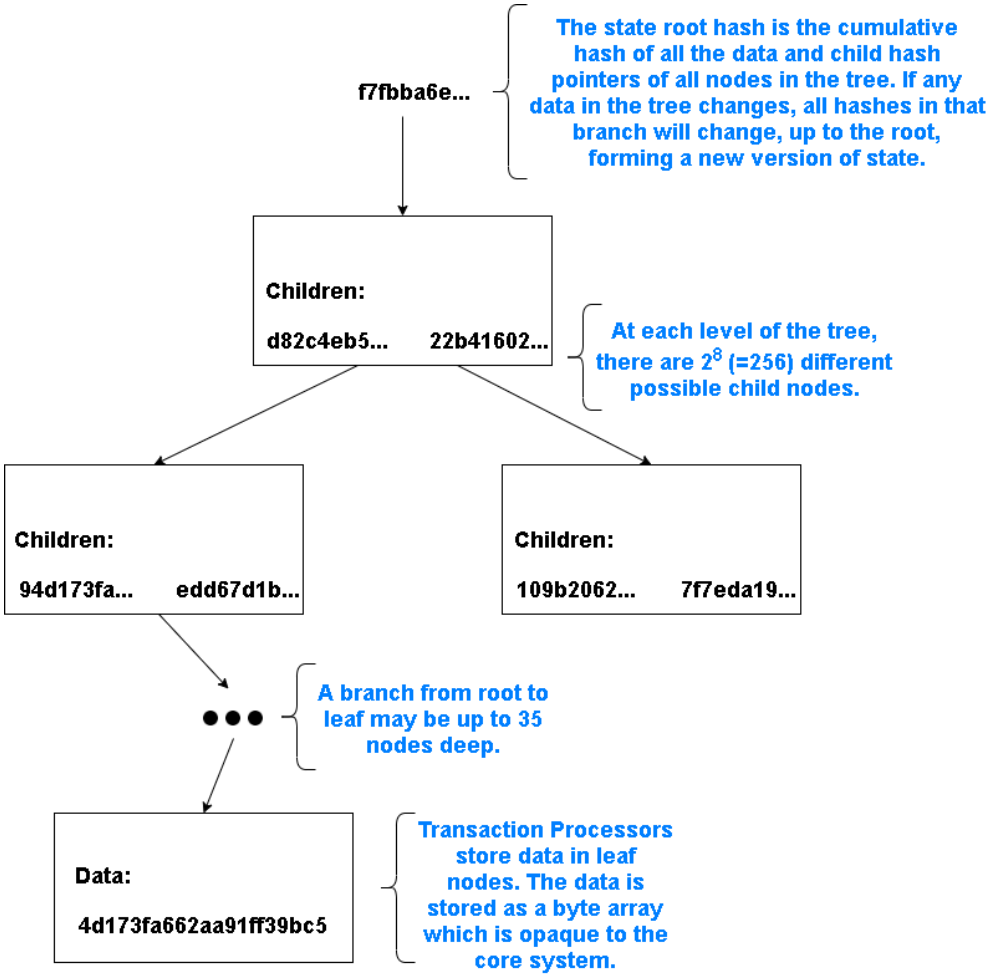


Figure 6.2.2: Merkle tree of the global state

It is denoted as an addressable Merkle Radix tree because addresses only identify the path to the leaf node. Figure 6.2.3 gives an example of an address. This address is 70 hexadecimal characters long which represents 35 bytes. Each byte corresponds to the choice of a child node, this is why each node can have $2^8 = 256$ child nodes. Given that there are 35 bytes, the tree can have a depth of 35 nodes. In this address, the 3 first bytes are reserved for the transaction family namespace. There are so $2^{3*8} = 2^{24} = 16777216$ possible different namespaces in a Sawtooth state.

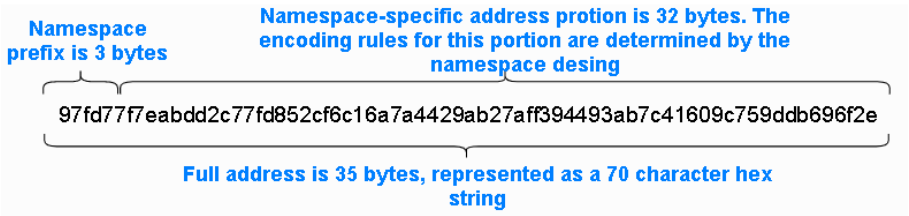


Figure 6.2.3: Example of address

The figure 6.2.4 shows the advantage to use a radix tree. Given that most of the time a transaction modifies only a small part of the state, the radix tree allows to reuse the part that is unchanged. Moreover, most of the addresses have common prefix. For example, all the data from the same transaction family have the same 3 bytes prefix. The radix tree uses this particularity to store the data in a more compact way.

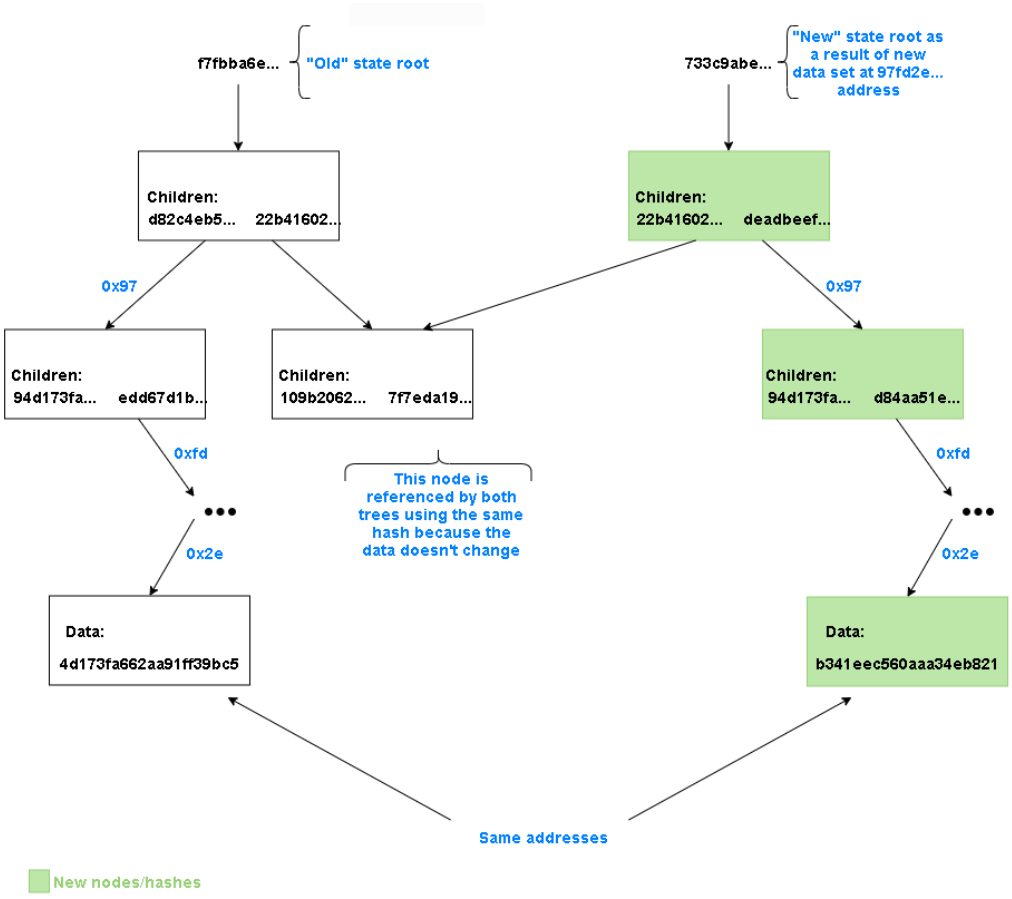


Figure 6.2.4: Radix tree of the global state

Note that there are some other concrete examples of addressing in the section 6.2.6 for specific transaction families.

6.2.3 Blocks, transactions and batches

Modifications to state are performed thanks to transactions. A client creates a transaction and submits it to the validator. Then, the validator applies the transaction which causes a change to the state. Transactions are always wrapped inside of a batch. In a batch, either all transactions are submitted or no transaction is submitted. The batch is the atomic unit of state change. Batches are then wrapped into blocks. The block, batch and transaction data structure are detailed in the annexes (cf. section 10.2).

The use of batches simplifies the dependency management between transactions. For example : there are three transactions A, B and C that should be apply in that order. However, if one of these transactions is invalid then no transaction should be apply. This problem is not solvable with only explicit dependencies between transaction. Imagine that transaction 'C' depends on transaction 'B', transaction 'B' depends on transaction 'A' and transaction 'A' depends on transaction 'C'. It is a cyclic dependency that can not be ordered.

It is possible to put transactions from different transaction families in the same batch. This encourages the reuse of transaction families.

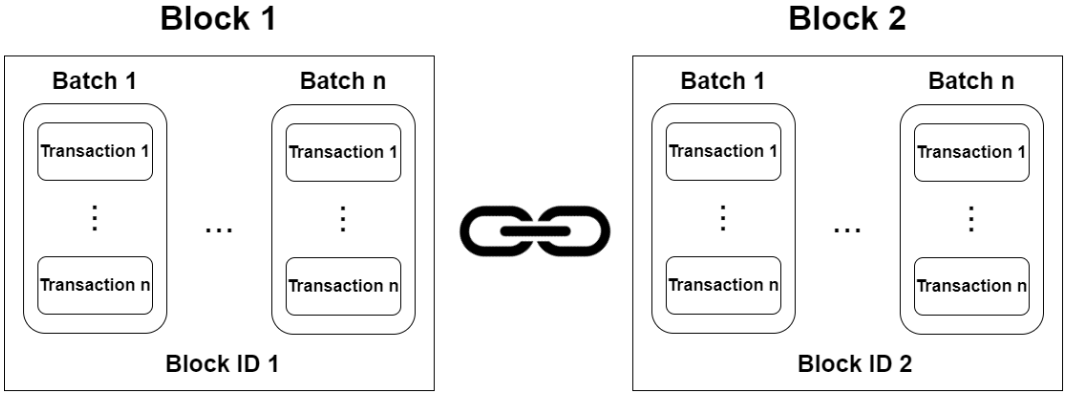


Figure 6.2.5: Relation between block, batch and transaction

6.2.4 Network layer security

This blockchain network includes a TLS-like certificate exchange mechanism and protocol encryption that is transparent to the socket implementation.

The TLS exchange mechanism enables a secure data transfer between two users. In the first part of the protocol, the two users exchange a symmetric key thanks to private-public key mechanism. In the second part of the exchange, data are encrypted and sent with the symmetric key obtained in the first part. The symmetric encryption guarantees the data confidentiality and an integrity check included in each transmitted message prevents loss or data alteration during transmission. Moreover, using asymmetric encryption which has a low throughput for the key exchange and the symmetric encryption which has a high throughput for the data exchange makes TLS highly efficient.

6.2.5 Network permissioning

The Sawtooth permissioning design is divided in two parts : the validator key permissioning and the transactor key permissioning. The organisation is the same for the two parts. There is a matching between a policy which is a set of nodes which follows the same rules and a role which defines what the policy is allowed to do. Nodes in the policy are identified by their public key. The set of roles and policies are implemented using the Identity Transaction Family. The different network and transaction roles are defined in the section 6.2.5.

The roles control the types of messages that can be sent and received over a given connection. The components and nodes that wish to take on these roles must participate in an authorization "handshake" (see section 6.2.5) and request for the roles that they want to take on. Then, based on the set of roles and identities stored within the identity namespace, the validator is able to determine whether messages delivered to it should be handled or dropped. For example, in the case a node is not allowed to connect to the network, the message is dropped and the connection is closed.
Sawtooth permissioning is set with on-chain or off-chain configuration. In on-chain configuration, settings are recorded in the blockchain, with changes or additions made as new blocks added to the blockchain. On-chain configuration applies to the entire Sawtooth network for that blockchain. In off-chain configuration, settings are recorded in the local *validator.toml* file, located by default at */etc/sawtooth/-validator.toml*, and applies only to the local validator node [59].

Validator key permissioning

The validator key permissioning controls which nodes are allowed to establish connection to the Sawtooth network. This setting is configured on chain which allows the whole network to change its policies at the same time while the network is live, instead of relying on a startup configuration.

Transactor key permissioning

The transactor key permissioning controls who can submit transactions and batches, based on signing key. Those settings are configured either on chain or off chain.

- Off-chain (local configuration) allows a validator to limit who is allowed to submit batches and transactions directly to that validator. This is useful, for example, if a company is running its own Sawtooth node and wishes to allow only members of that company to submit transactions. The validator only accepts batches and transactions from predefined transactors that are loaded from a local validator config file. Once the validator is configured, the list of allowed transactors is immutable while the validator is running.
- On-chain configuration uses the Identity namespace which handles network-wide updates and agreement on allowed transactors. The allowed transactors are checked when a batch is received from a client, received from a peer, and when a block is validated. The Identity and Settings transaction processor (or equivalent) are required to process the on-chain settings. Note that Settings includes several settings to control who can change on-chain settings and how many votes are required for a setting change.

Network roles

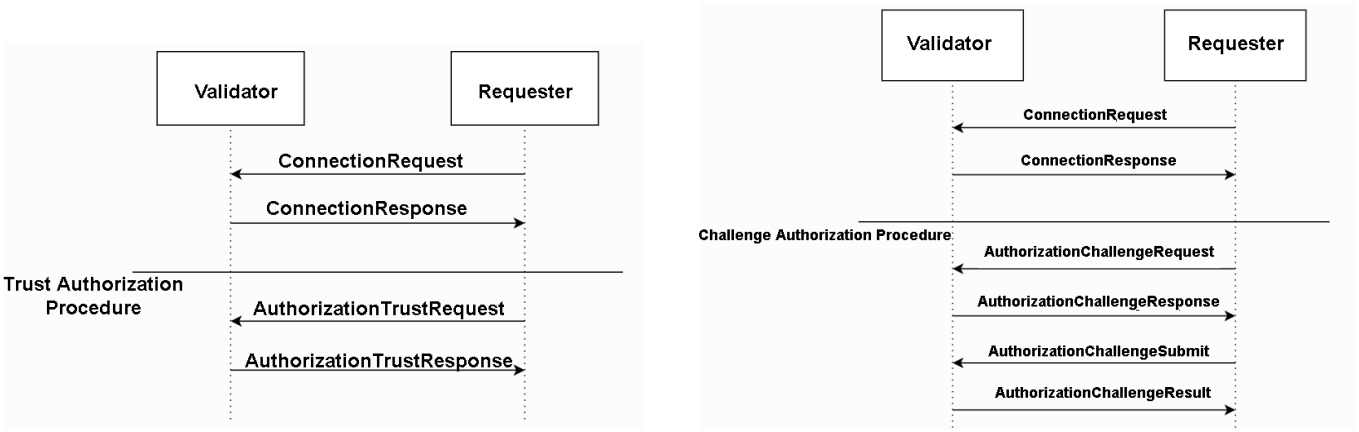
- **default** : if the role has not been set, the default policy is used. The default policy permits all the keys to be part of the network (PERMIT_KEY *).
- **network** : if a validator receives a peer request from a node whose validator public key is not permitted by the policy, the message will be dropped, an AuthorizationViolation will be returned, and the connection will be closed.
- **network.consensus** : if a validator receives a GossipMessage that contains a new block published by a node whose validator public key is not permitted by the policy, the message will be dropped, an AuthorizationViolation will be returned, and the connection will be closed.

Transaction roles

- **default** : if the role has not been set, the default policy is used. The default policy permits all the keys to sign transactions an batches (PERMIT_KEY *)

- **transactor** : the top level role for controlling who can sign transactions and batches on the system. This role is used when the allowed transactors for transactions and batches are the same. Any transactor whose public key is in the policy is allowed to sign transactions and batches, unless a more specific sub-role disallows their public key.
- **transactor.transaction_signer** : if a transaction is received that is signed by a transactor who is not permitted by the policy, the batch containing the transaction will be dropped.
- **transactor.transaction_signer.{tp_name}** : if a transaction is received for a specific transaction family that is signed by a transactor who is not permitted by the policy, the batch containing the transaction will be dropped. {tp_name} refers to a transaction family name
- **transactor.batch_signer** : if a received batch is signed by a transactor that is not permitted by the policy, that batch will be dropped.

Handshake procedure



a) authorization trust b) authorization challenge

Figure 6.2.6

When a new node (the requester) wants to join the network, it has to open a connection with the validator of a node already present in the blockchain. To do so there are two possible handshake procedures represented in the figure 6.2.6. The first step is common for the two. The node that wants to connect to the validator sends a *connectionRequest* message. The validator answers with a *connectionResponse* message. This message contains a field *authorizationType* which describes the

authorization procedure and a field *roleEntry* that describes the type of connections that are supported on the endpoint. The "trust authorization" is the simplest case. The validator trusts the public key of the requester and gives it the requested role if it is available. The "challenge authorization" is more complex because the validator does not trust the requester public key. The requester has to proof its identity by signing a challenge with its private key. A more complete explanation of the different procedures and the structure of the messages are presented in the annexes (cf. section 10.4).

Example of network permissioning

In order to illustrate the network permissioning, an example of a concrete case is illustrated in the figure 6.2.7. As represented at the output of the first command, there are two policies named **policy_batch** and **policy_network**. A policy is a set of PERMIT_KEY and DENY_KEY rules that are evaluated in the order listed. In our example, there are no DENY_KEY since all the keys that are not in the PERMIT_KEY are by default in the DENY_KEY. As represented at the last two lines of the figure 6.2.7, each policy is associated with a particular role. A role is an authorization that grants permission to perform operations and access data. A policy associated with the **network** role means that those nodes will only connect to the nodes related to the policy. If a node tries to connect to the network but has an ID that is not in the policy **policy_network**, then the other nodes of the network will reject the connection and a *AthorizationViolation* message is output (see section 6.2.5). On the other hand, the policy **policy_batch** is associated with the **transactor** role. It means that the nodes of the policy **policy_batch** are the only ones authorized to sign transactions and batches.
Note that the ID's of the **policy_batch** and **policy_network** refer to the same nodes. Indeed, the policies associated with a network role must contain the validator public keys of the node (contained in *etc/sawtooth/keys/validator.pub*). On the other hand, the policies associated with a transaction role must contain the transactor public key of the node (contained in *home/yourname/.sawtooth/keys/my_key.pub*). In this example, the transactor public key of the first node is 033cc9c1... while its validator public key is 03753418d9...

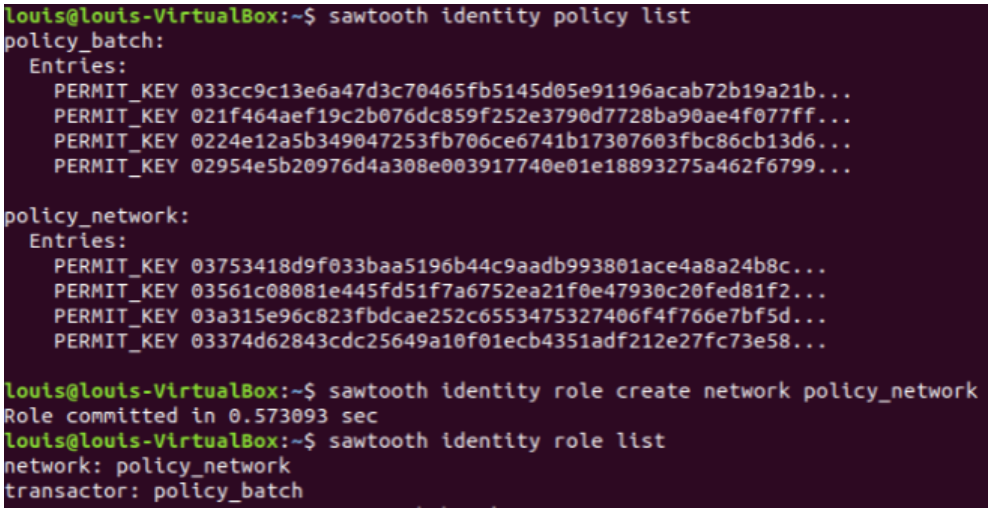


Figure 6.2.7: Example of network permissioning

6.2.6 Transaction family specifications

One of the advantages of Sawtooth is the separation between the application level and the core system. This is possible because each application defines its own transaction family adapted to its requirements. A transaction family includes these components :

- A transaction processor to define the business logic of the application.
- A data model to record and store data.
- A client to handle the client logic of the application.

Several transaction families are included in the sawtooth-core repository and can be reused to develop new projects.

Identity transaction family

Overview This transaction family is used to manage the on chain transactor key and validator key permissioning. This requires to store the list of authorized participants for each role. Currently, all the participants in the network are identified by their public key. However, this is difficult to manage. An identity namespace is so used to streamline managing identities.

The identity namespace :

- Encompasses ways to identify participants based on public keys.

- Stores a set of nodes that follow the same rules called "policies".
- Stores the roles that those policies apply to.

State : Policies A policy has a unique name and contains a list of entries. Each entry corresponds to a participant and contains the public key of this participant and the type of entry (if it is a permit or deny key).

```
// Policy will be stored in a PolicyList to account for hash collisions
message PolicyList {
  repeated Policy policies = 1;
}

message Policy {

  enum EntryType {
    ENTRY_TYPE_UNSET = 0;
    PERMIT_KEY = 1;
    DENY_KEY = 2;
  }

  message Entry {
    // Whether this is a PERMIT_KEY or DENY_KEY entry
    EntryType type = 1;

    // This should a public key or * to refer to all participants.
    string key = 2;
  }

  // name of the policy, this should be unique.
  string name = 1;

  // list of Entries
  // The entries will be processed in order from first to last.
  repeated Entry entries = 2;
}
```

Figure 6.2.8

State : Roles A role is the mapping between a role and a policy. It contains the role name and the policy name.

```
// Roles will be stored in a RoleList to handle state collisions. The Roles
// will be sorted by name.
message RoleList {
  repeated Role roles = 1;
}

message Role{
  // Role name
  string name = 1;

  // Name of corresponding policy
  string policy_name = 2;
}
```

Figure 6.2.9

Transaction Payload The figure 6.2.10 explicits the payload format.

```
message IdentityPayload {
    enum IdentityType {
      POLICY = 0;
      ROLE = 1;
    }

    // Which type of payload this is for
    IdentityType type = 1;

    // Serialize bytes of a role or a policy
    bytes data = 2;
}
```

Figure 6.2.10

Addressing All identity data are stored under the special namespace "00001d" The address for policies are formed with the special policy namespace "00" concatenated with the 62 first characters of the SHA256 hash of the policy name.

$$'00001d' +' 00' + hashlib.sha256('policy\_name'.encode()).hexdigest()[:62]$$

The address for roles is formed with the concatenation of four parts delimited by the "." character. If there is less than four parts, the last part is filled with the empty string. For example a.b.c is split into "a", "b", "c" and the empty string. If there are more than four parts, the extra part is left in the last part. For example a.b.c.d.e is split into "a", "b", "c" and "d.e". For each part the SHA-256 hash is computed and the 16 first characters are conserved.
The address for role is formed with the special role namespace "01" and the four short hashes. For example, the address for the role "transactor.transaction_signer"

is built as follows:

$$
\begin{aligned}
'00001d' +' 01' &+ hashlib.sha256('transactor'.encode()).hexdigest()[:14] \\
&+ hashlib.sha256('transactor\_signer'.encode()).hexdigest()[:16] \\
&+ hashlib.sha256(''.encode()).hexdigest()[:16] \\
&+ hashlib.sha256(''.encode()).hexdigest()[:16]
\end{aligned}
$$

Settings transaction family

Overview This transaction family provides a methodology for storing on-chain configuration settings. The settings stored in state as a result of this transaction family play a critical role in the operation of a validator. This design supports two authorization options: a single authorized key that can make changes and multiple authorized keys. In the case of multiple keys, a configuration setting controls how many nodes must vote to make a change.

State Settings data consists of a pair settings_name - value. The table 6.2.11 shows some example of settings that are stored in the transaction family. The table 6.2.12 shows the settings that the transaction family uses for its own configuration.

Setting(Examples)	**Value**
sawtooth.poet.target_wait_time	5
sawtooth.validator.max_transactions_per_block	1000

Figure 6.2.11: Example of settings

Settings(Settings)	**Value Description**
sawtooth.settings.vote.authorized_key	List of public key allowed to propose and vote on settings changes
sawtooth.settings.vote.approval_threashold	Minimum number of votes required to accept or reject a proposal
sawtooth.settings.vote.proposals	A list of proposal to make settings changes

Figure 6.2.12: Family transaction settings

Addressing All settings data are stored under the special namespace "000000". As for the Identity transaction family, the addresses are split into four parts delimited by a dot character. A SHA-256 hash of each part is computed and

the 16 first characters are conserved. For example the address of the setting : sawtooth.settings.vote.proposals could be set like this :

$$
\begin{aligned}
'000000' + hashlib.sha256('sawtooth'.encode()).hexdigest()[:16] \\
+ hashlib.sha256('settings'.encode()).hexdigest()[:16] \\
+ hashlib.sha256('vote'.encode()).hexdigest()[:16] \\
+ hashlib.sha256('proposals'.encode()).hexdigest()[:16]
\end{aligned}
$$

Transaction payload the figure 6.2.13 explicits the payload format.

```
// Setting Payload
// - Contains either a proposal or a vote.
message SettingPayload {
    // The action indicates data is contained within this payload
    enum Action {
        // A proposal action - data will be a SettingProposal
        PROPOSE = 0;

        // A vote action - data will be a SettingVote
        VOTE = 1;
    }
    // The action of this payload
    Action action = 1;

    // The content of this payload
    bytes data = 2;
}

// Setting Proposal
//
// This message proposes a change in a setting value.
message SettingProposal {
    // The setting key.  E.g. sawtooth.consensus.module
    string setting = 1;

    // The setting value. E.g. 'poet'
    string value = 2;

    // allow duplicate proposals with different hashes
    // randomly created by the client
    string nonce = 3;
}

// Setting Vote
//
// In ballot mode, a proposal must be voted on.  This message indicates an
// acceptance or rejection of a proposal, where the proposal is identified
// by its id.
message SettingVote {
    enum Vote {
        ACCEPT = 0;
        REJECT = 1;
    }

    // The id of the proposal, as found in the
    // sawtooth.settings.vote.proposals setting field
    string proposal_id = 1;

    Vote vote = 2;
}
```

Figure 6.2.13

6.2.7 Consensus algorithm

Sawtooth abstracts the core concepts of consensus and isolates consensus from transaction semantics. The Sawtooth consensus interface supports plugging in various consensus implementations as consensus engines that interact with the validator through the consensus API. More importantly, Sawtooth allows to change the consensus after the blockchain network has been created. The consensus

algorithm is selected during the initial network setup and can be changed on a running blockchain. The Sawtooth consensus API supports a wide variety of consensus algorithms on a network. Sawtooth currently includes consensus engines for these algorithms [60]:

1. Sawtooth PBFT (Practical Byzantine Fault Tolerance) is a voting-based consensus algorithm that provides Byzantine fault tolerance with finality. The advantage of the PBFT consensus algorithm is that it tolerates faults of a certain number of Byzantine nodes. However, PBFT features complex communication and low scalability. When the number of consensus nodes in the distributed system is large enough, the functionality will go down dramatically. When there is a great deal of network partition, consensus efficiency will be compromised and result in a huge delay.

2. PoET (Proof of Elapsed Time) is a Nakamoto-style consensus algorithm that is designed to be a production-grade protocol capable of supporting large network populations. PoET relies on secure instruction execution to achieve the scaling benefits of a Nakamoto-style consensus algorithm without the power consumption drawbacks of the proof of work algorithm. Sawtooth includes two versions of PoET consensus:

    - PoET-SGX relies on a Trusted Execution Environment (TEE), such as Intel Software Guard Extensions (SGX), to implement a leader-election lottery system. PoET-SGX is sometimes called "PoET/BFT" because it is Byzantine fault tolerant.
    - PoET simulator provides PoET-style consensus on any type of hardware, including a virtualized cloud environment. PoET simulator is also called "PoET/CFT" because it is crash fault tolerant, not Byzantine fault tolerant (see section 9.3).

3. Sawtooth Raft is a leader-based consensus algorithm that provides crash fault tolerance (see section 9.3) for a small network with restricted membership.

4. Devmode (short for "developer mode") is a simplified random-leader algorithm that is useful for developing and testing a transaction processor. Devmode is not recommended for multi-node networks and should not be used for production.

Sawtooth PBFT [79], [80]

The Sawtooth PBFT algorithm is a voting-based consensus algorithm with Byzantine fault tolerance. The network can tolerate a certain number of faulty

nodes. As long as this number is not exceeded, the network will work properly. More precisely, the network can work properly if the number of faulty nodes is smaller than $\frac{n-1}{3}$ with n, the total number of nodes in the network.
There are two types of nodes in this kind of network: the primary node and the secondary nodes. The primary node builds and publishes blocks. The secondary nodes vote on blocks and the health of the leader. A view is the period of time that a given node is the primary. A view change means switching to a different primary node. The next primary is selected in a round-robin (circular) fashion. The Sawtooth PBFT algorithm changes the primary at regular intervals, as well as when the secondary nodes determine that the current primary is faulty.
In Sawtooth PBFT, each node stores several key pieces of information as part of its state such as:

- List of PBFT member nodes in the network.
- Current sequence number, which is also the number of the block being processed.
- Current view number, which identifies the primary node.
- The current head of the chain.
- Log of all blocks it has received.
- Log of all messages it has received.

Sawtooth PBFT also configures the network with on-chain settings. These settings list each node in the network, set the view-change interval (how often the primary changes), and specify other items such as the block publishing frequency, timeout periods, and message log size.

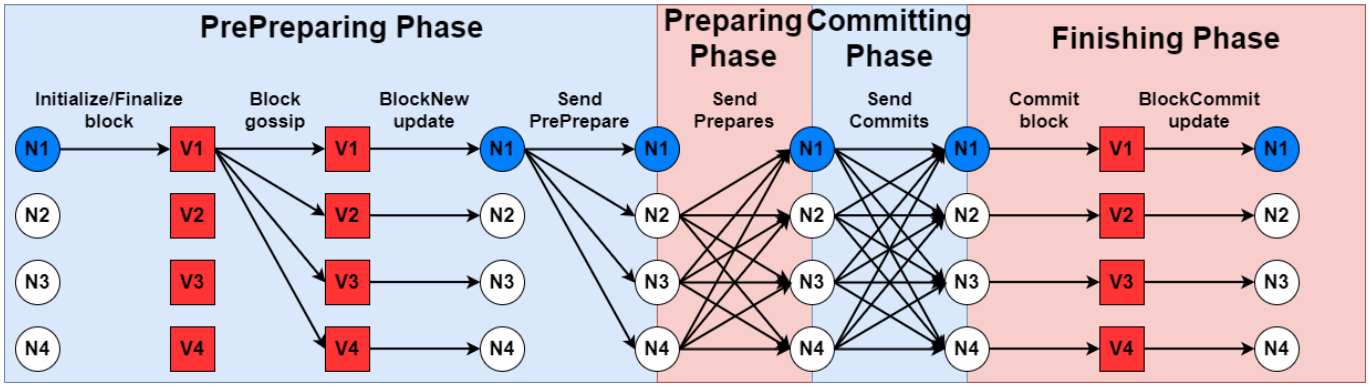


Figure 6.2.14: Graphical representation of the Sawtooth PBFT in the normal case [71]

As represented in the figure 6.2.14, there are different types of messages. When a node receives such a message, it performs a series of checks to ensure the validity of the message:

1. The signature of the message is verified.
2. The message must be from a known PBFT member.

The first check verifies that the PBFT message was created and signed by the node that it claims to be from. This guarantees the origin of the PBFT message.
The second check ensures that only nodes that are accepted members of the PBFT network are able to participate in the consensus process.
Any messages that fail to parse or pass the required checks are ignored. If a message is successfully parsed and passes verification, it is passed to a handler for that specific message type, where it may go through further checks, be stored in the message log, or trigger some actions.

PBFT operation in the normal mode

Let's now analyse deeper the consensus operation in the normal mode (when there is no view change). This is represented in the figure 6.2.14. In this figure, N1 is the primary node and N2, N3, and N4 are secondary nodes. The normal mode proceeds as follows:

1. All nodes begin in the *PrePreparing* phase; the purpose of this phase is for the primary to publish a new block and endorse the block with a *PrePrepare* message.
    - The primary node will send a request to its validator to initialize a new block. After a configurable timeout, the primary will send a request to the validator to finalize the block and broadcast it to the network.
    - After receiving the block in a *BlockNew* update and ensuring that the block is valid, all nodes will store the block in their PBFT logs.
    - After receiving the *BlockNew* update, the primary will broadcast a *PrePrepare* message for that block to all of the nodes in the network. When the nodes receive this *PrePrepare* message, they will make sure it is valid. If it is, they will add it to their respective logs and move on to the *Preparing* phase.
2. In the *Preparing* phase, all secondary nodes (not the primary) broadcast a *Prepare* message that matches the accepted *PrePrepare* message. Each node will then add its own *Prepare* to its log, then accept *Prepare* messages from

other nodes and add them to its log. Once a node has 2f *Prepare* messages in its log that match the accepted *PrePrepare*, it will move on to the *Committing* phase.

3. The *Committing* phase is similar to the *Preparing* phase; nodes broadcast a *Commit* message to all nodes in the network, wait until there are 2f + 1 *Commit* messages in their logs, then move on to the *Finishing* phase. The only major difference between the *Preparing* and *Committing* phases is that in the *Committing* phase, the primary node is allowed to broadcast a message.

4. Once in the *Finishing* phase, each node will tell its validator to commit the block for which they have a matching *PrePrepare*, 2f *Prepare* messages, and 2f + 1 *Commit* messages. The node will then wait for a *BlockCommit* notification from its validator to signal that the block has been successfully committed to the chain. After receiving this confirmation, the node will update its state as follows:

   - Increment its sequence number by 1.
   - Update its current chain head to the block that was just committed.
   - Reset its phase to *PrePreparing*.

   Finally, the primary node will initialize a new block to start the process all over again.

View change

Up to know the procedure was explained in a case where the primary node is considered valid. However, a node starts a new view change if any of the following occur:

- The idle timeout expires: when a node enters the *PrePreparing* phase, it will start its idle timeout. If the node receives a new block and a matching *PrePrepare* from the primary for its current sequence number before the timeout expires, it will stop the timeout; if not, the node will initiate a view change when the timeout expires.

- The commit timeout expires: when a node enters the *Preparing* phase, it will start its commit timeout. If the node is able to move on to the *Finishing* phase and send a request to the validator to commit the block before the timeout expires, it will stop the timeout; if not, the node will initiate a view change when the timeout expires.

- The view change timeout expires: when a node starts a view change to view 'v', it will start a view change timeout. If the node is able to complete the view change before the timeout expires, it will stop the timeout; if not, it will initiate a new view change to view 'v + 1'.
- Multiple *PrePrepare* messages are received for the same view and sequence number but different blocks: this indicates the primary is faulty, since this behavior is invalid.
- A *Prepare* message is received from the primary: this indicates the primary is faulty, since this behavior is invalid.
- 'f + 1' *ViewChange* messages are received for the same view: this ensures that a node does not wait too long to start a view change; since only 'f' nodes (at most) can be faulty at any given time, if more than 'f' nodes decide to start a view change, other nodes can safely join them to perform that view change.

To start a view change, a node will do the following:

1. Update its mode to *ViewChanging(v)*, where 'v' is the view the node is changing to.
2. Stop both the idle and and commit timeouts, since these are not needed again until after the view change.
3. Stop the view change timeout if it's been started; it will be restarted with a new value later.
4. Broadcast a *ViewChange* message for the new view.

ViewChange messages are accepted and added to the log if they satisfy these criteria:

- They are for a later view than the node's current view.
- If the node is in the mode *ViewChanging(v)*, the view in the message must be greater than or equal to 'v'.

Once a node has received '2f + 1' *ViewChange* messages for the new view, it will start its view change timeout.
When the primary for the new view receives '2f + 1' *ViewChange* messages, it will broadcast a *NewView* message to the network, signifying that the view change is complete. As a proof that this view change is valid, the primary will include '2f + 1' signed *ViewChange* messages from other nodes in the *NewView* message (the primary's own "vote" is implicit), which will be validated by the other nodes.
If a node receives the new primary's valid *NewView* message before its view change timeout expires, it will:

1. Stop the view change timeout.

2. Update its view to match the new value.

3. Revert back to Normal mode.

However, if a node's view change timeout expires before it receives a *NewView*, it will stop the timeout and initiate a brand new view change for view 'v + 1' (where 'v' is the view it was attempting to change to before).

Role of the different phases

According to the original paper of M. Castro et al. describing the PBFT [1], the pre-prepare and prepare phases are used to totally order requests sent in the same view, even when the primary, which proposes the ordering of requests, is faulty. The prepare and commit phases are used to ensure that requests that commit are totally ordered across views. Indeed, replicas (nodes) may collect the 2f *Prepare* messages in different views with the same sequence number and different requests. The commit phase solves this problem as follows. Each replica i multicasts *Commit* messages saying it received 2f *Prepare* messages and adds this message to its log. Let's illustrate the importance of the *Committing* phase with an example: imagine that the *Committing* phase does not exist. Due to network delay, a node X is the only node that directly receives 2f+1 *Prepare* messages. Because there is no *Committing* phase, node X directly commits the block (let's call it B1). Now a changing of view happens and a new primary node will submit a new transaction with the old sequence number (since node X was the only node that received enough *Prepare* messages). If, due to network delay, all the nodes receive 2f+1 *Prepare* messages for this new transaction excepted node X, then all those nodes will commit the new block (called B2). In this case, the total order of the committed blocks is not respected. Indeed, committed block order of node X is B1 and then B2 while the committed block order of the other nodes is B2 and then B1.
Thanks to the *Committing* phase, this kind of behavior can not happen. Indeed, with this phase, if the node X is the first node that receives the 2f+1 *Commit* messages, then it knows that the other nodes already received 2f+1 *Prepare* messages and that a quorum accepted to assign a particular sequence number to a request. Thanks to that, this sequence number will not be re-used even in a new view so the total order of the committed blocks will be respected.

6.3 Our Sawtooth implementation

6.3.1 Network permissioning

As mentioned in the chapter 5, the blockchain should be private. To restrict the access to the network, a validator key permissioning is set. The policy contains the validator public key (*etc/sawtooth/keys/validator.pub*) of all the participants of the network. The role associated to this policy is "network". In this way if a node absent of the policy tries to connect to the network, it receives an "authorizationViolation" message and the connection is closed.

A transactor key permissioning is set to restrict the nodes allowed to commit new transactions. In the integration of our solution in the WeSmart implementation, the only block that submit new transactions is the WeSmart node. The policy contains only the transactor public key of the WeSmart node (*home/yourname/.sawtooth/keys/my_key.pub*). The role associated with this policy is "transactor". In this way, the WeSmart node is the only node who can sign transactions and batches on the network.
Moreover, a TLS-like certificate exchange mechanism and protocol encryption is implemented for the communication between nodes. This is done at the creation of the node with public and private network key set in the *validator.toml* file (the complete procedure is described in the annexes at the section 11.6).

6.3.2 WE transaction family

Overview This transaction family enables to store the hash of the energy consumption associated to a certain timestamp. Then, given this timestamp, it is possible to recover the hash of the associated energy consumption.

State The WE state entry consists of a UTF-8 encoding string with exactly two "-" characters which is formatted as follows :

$$< date > - < listId > - < listConsumption >$$

- date : represents the timestamps corresponding to the energy consumption.
  Format : year:month:day:hour(24):min
- listId : represents the hash of a list containing the identifier of the consumers.
  Format : hash(id1,id2,id3...)
- listConsumption : represents the hash of a list containing the consumption of the consumers.
  Format : hash(consumption1,consumption2,consumption3...)

Addressing We data are stored in state using addresses generated form the WE family name and the date of the energy consumption. The namespace corresponding to the WE transaction family consists of the 6 first characters of the SHA-512 hash of the UTF-8 encoding of the string "we". The next 64 characters of the address are the 64 first characters of the SHA-256 hash of the UTF-8 encoding of the date. For example the WE address of a state with date "2021:04:12:15:35" could be generated as follow :

$$hashlib.sha512('we'.encode('utf-8')).hexdigest()[:6] +$$
$$hashlib.sha512('2021:04:12:15:35'.encode('utf-8')).hexdigest()[:64]$$

Transaction payload A WE transaction request payload consists of a UTF-8 encoding string with exactly three "-" characters which is format as follows :

$$< date > - < action > - < listId > - < listConsumption >$$

- date : represents the timestamp of the energy consumption.
- action : indicates the type of actions to perform; *set* to add data to the blockchain or *get* to recover data from the blockchain.
- listId : (not required if *get*) represents the hash of a list containing the identifier of the consumers. Format : hash(id1,id2,id3...)
- listConsumption : (not required if *get*) represents the hash of a list containing the consumption of the consumers.
  Format : hash(consumption1,consumption2,consumption3...)

Execution The interaction with the blockchain is performed by two commands :

- we set -date -listId -listConsumption
- we get -date

The *set* command allows the user to add data in the blockchain. *-listId* and *-listConsumption* are the hashes of the list of the user and the list of the corresponding consumption. The *-date* corresponds to the period relative to the consumption. In practice, a new block is created with an updated Merkle tree. The hash of *listId* and *listconsumption* are stored in the address generated by the date (cf. section addressing).

The *get* command allows the user to ask for data in the blockchain. *-date* corresponds to the period relative to the consumption. In practice, this command

computes the address where data are stored in the Merkle tree thanks to the date (cf. section addressing). Then, it returns the data by searching at this address in the Merkle tree of the current blockchain head block.

Global working

This section describes the flow of a block creation following the figure 6.3.1. This figure introduces the journal which is the group of validator subcomponents that works together to handle batches and proposed blocks. These components are responsible for completing published blocks, publishing batches into blocks to extend the chain and validating proposed blocks to determine if they should be considered for the new chain head. Here is a brief description of the journal architecture. A detailed explanation of every block is presented in the annexes (cf. section 10.3).
Blocks and batches arrive via interconnect, either through the gossip protocol or from client requests. Then, the processing of these blocks and batches is handled in multiple pipelines :

- The Completer initially receives the blocks and batches. It guarantees that all dependencies for the blocks and batches have been satisfied.
- Completed batches go to the BlockPublisher for batch validation and inclusion in a block.
- Completed blocks go to the ChainController for block validation and fork resolution.
- The BlockCache and BlockStore provide storage for the batches and blocks being processed.

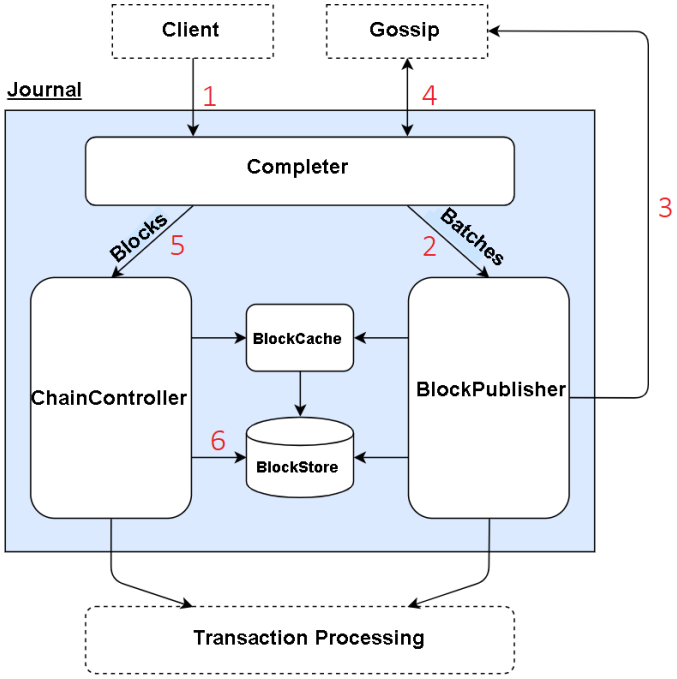


Figure 6.3.1: Journal

In our implementation, the workflow is the following. When a client wants to perform a *set*, a transaction is created in the *we_client* file. The format of this transaction is explained in the previous section. The payload of the transaction has the format "date-action-listId-listConsumption". This transaction is then put into a batch. Since, in our work, there are no transaction dependencies, there is a single transaction per batch. This batch is then sent to the validator thanks to a POST message through the rest API.

In the validator, the proposed batch is received by the Completer (1 in the figure 6.3.1). The Completer checks that the dependencies between the transactions are satisfied. In our implementation, it is always the case since there are no dependencies between transactions (no transaction should be posted before an other).

When a batch is considered "completed" by the Completer, it goes to the BlockPublisher (2 in the figure 6.3.1). Its role is to create a candidate block to extend the current chain. The BlockPublisher interacts with the transaction processor to compute the *state_root_hash* field in the block header. Indeed, this is the root of the Merkle tree after all transactions of the block have been applied on the global state. The way the addressing of the Merkle tree is defined depends on the transaction family specification (cf.WE transaction family). The BlockPublisher interacts also with the consensus engine to validate and publish

a block. Following the instructions of the consensus engine, the BlockPublisher broadcasts the proposed block to the other nodes of the network (3 in the figure 6.3.1).

Once the proposed block is published by the Blockpublisher, all the nodes receive the block in the Completer which checks that the block is complete (4 in the figure 6.3.1). This means that all the predecessor block have been delivered to the Chaincontroller. It also checks that the batch field in the proposed block contains all the batches and in the same order than specified in the block header.

Once a block is considered complete by the Completer, it is delivered to the ChainControler for validation (5 in the figure 6.3.1). The role of the ChainControler is to maintain the current chain head of the blockchain for the validator. This responsibility involves validating proposed blocks, evaluating valid blocks to determine if they should be considered for the new chain head, and generating new blocks to extend the chain. The ChainController creates a scheduler to compute new state with a related Merkle hash for the block being published. The Merkle hash is compared to the state root contained in the block header. If they match, the block is valid from a transaction execution and state standpoint. The ChainController uses this information in combination with consensus information to determine whether to update the current chain head. When a block has been validated by $\frac{2n}{3} + 1$ nodes (see section 6.2.7) the block is added to the blockchain.

When a block is validated by the ChainController and so added to the blockchain, it is stored in the BlockStore which is a persistant on-disk storage of all blocks in the current chain head back to the genesis (6 in the figure 6.3.1).

The working set of blocks for the validator is stored in the BlockCache.

Chapter 7

Performance analysis

This section presents the performances of the Sawtooth blockchain. The Sawtooth statistics are generated thanks to InfluxDB, Grafana and Telegraf. The metrics gathering data flow is shown in the figure 7.0.1.

In the left of the figure, Sawtooth validator and Rest API send metrics to InfluxDB which is a time series database that is optimized for fast access to time series data. Moreover Telegraf which is a metrics reporting agent, gathers supplemental system information from the Linux kernel. Data are then read from InfluxDB by Grafana which is a multi-platform open source analytics and interactive web application. Finally Grafana displays several graphics in the web browser that can be interpreted by the user.

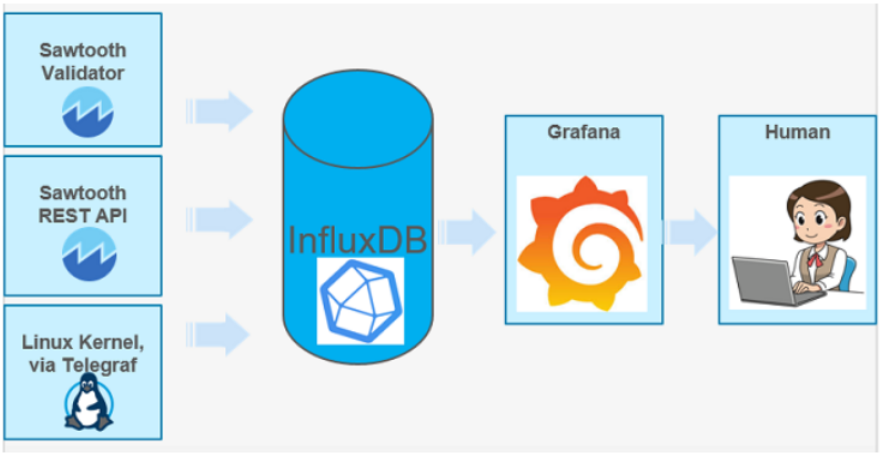


Figure 7.0.1: Metrics gathering data flow [37]

The figure 7.0.2 represents several graphics that are obtained with Grafana. In

this execution, the rate at which blocks are added to the blockchain is increased step by step. The graph a) represents the number of committed transactions that increases and the graph b) represents the rate at which blocks are added. The blockchain is able to tolerate up to 4 batches per second. At a rate of 5 batches per second, the throughput limit of the blockchain is reached. At this high rate, a protection mechanism for the blockchain gets ready. This is the backpressure test and its role is to prevent DoS (Deny of Service) attacks. If the validator is overwhelmed, it stops accepting new batches and the Rest API sends an error "429 : Too Many Requests" to the client. The number of batches that validator can accept is based on a multiplier, QUEUE_MULTIPLIER, times a rolling average of the number of published batches.
The graph c) represents the number of backpressure batches exchanged in the blockchain. The graphs d) displays statistics of the node on which the blockchain is deployed. In this particular case, the node is an ubuntu (64 bits) virtual machine with 2 processors and 2816 MB of RAM.

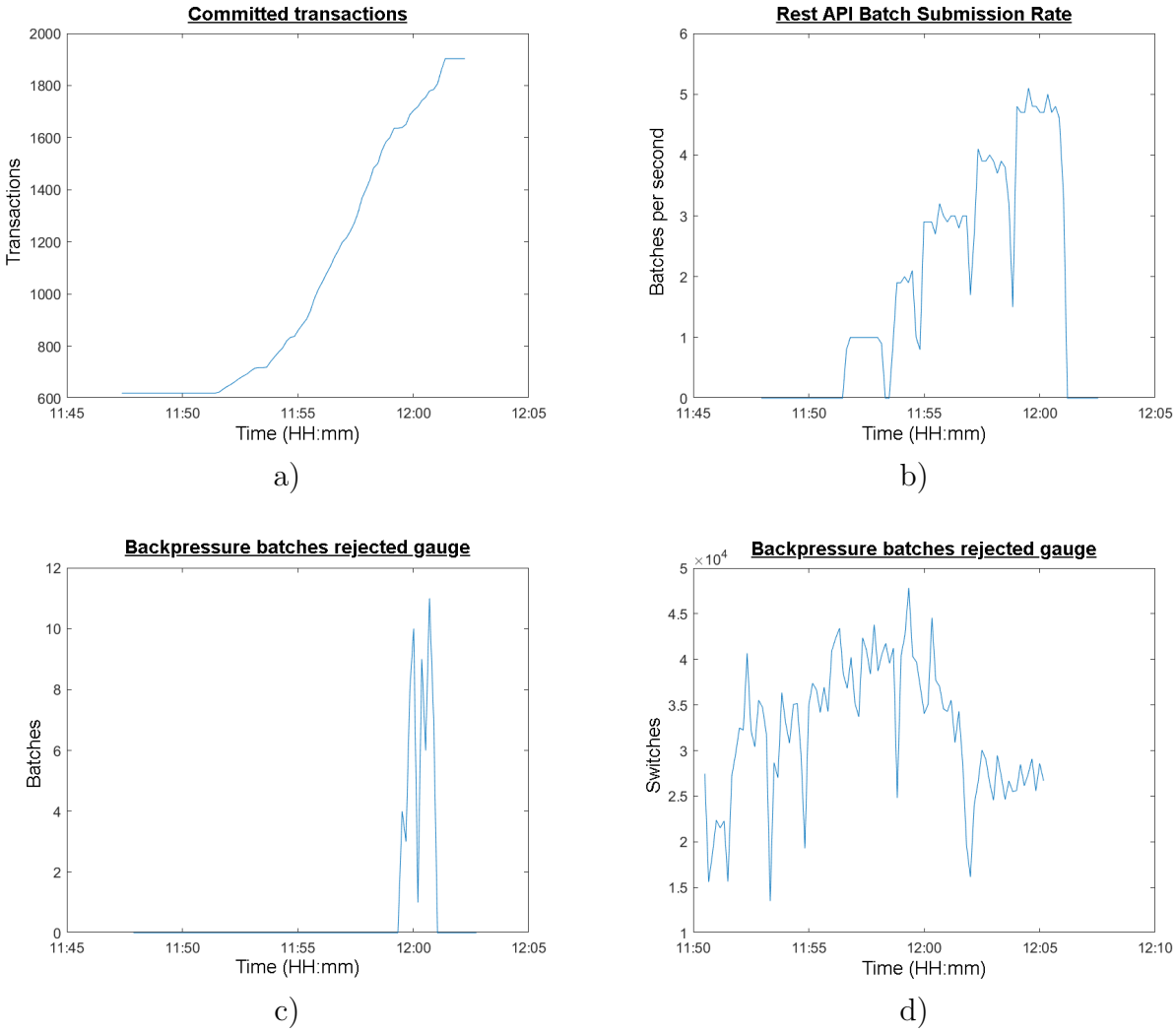


Figure 7.0.2

In the Sawtooth implementation, the blockchain is stored in the folder *var/lib/-sawtooth*. The blockchain is stored in a lightning memory-mapped database (LMDB). The figure 7.0.3 represents the size of the database depending on the number of stored blocks. In this experimentation, the blocks are generated with the WE transaction family. It is so representative of the solution that can be deployed by WeSmart. An important observation is that the size of the blockchain increases linearly with the number of blocks. Given that one block is generated every 15 minutes, 96 blocks are generated everyday. Every month 3000 blocks are generated which corresponds to 42.3MB of storage. An extrapolation of this graph leads to a storage consumption of 508MB per year.

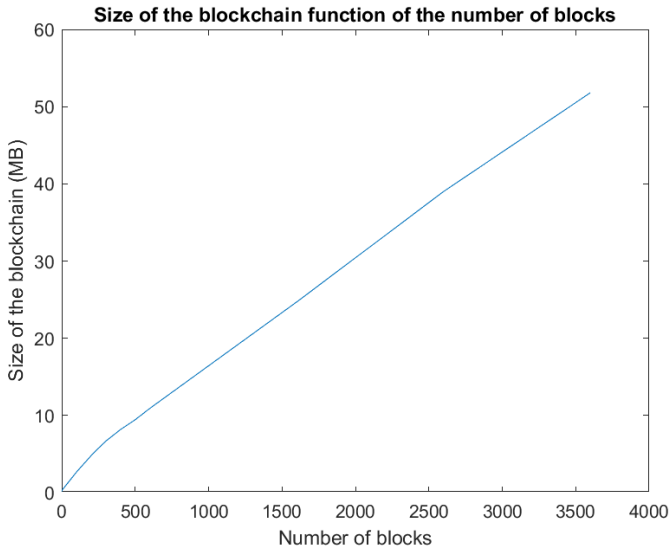

Size of the blockchain function of the number of blocks
Size of the blockchain (MB)
Number of blocks
0
10
20
30
40
50
60
0
500
1000
1500
2000
2500
3000
3500
4000

Figure 7.0.3

Chapter 8

Future work

In this work, we implemented a blockchain-based mechanism to ensure the integrity of the data stored in the cloud. The first part of this section proposes different possible improvements that can be done in the future by the WeSmart company. The second part introduces the concept of hierarchical blockchain which can solve some limitations of common blockchain.

8.1 Improvement of our solution

Up to now, the blockchain has been used to ensure the integrity of the data stored in the cloud. However, as explained in the chapter 3, the blockchain could also be used as the central part of the energy trading. As in the Brooklyn microgrid project (section 3.2), the blockchain could be used to implement the information system but also the market and pricing mechanism. In this example, smart meters send the energy generation and demand to a smart contract. Its role is to dispatch the energy according to rules that have been defined at the creation of the blockchain. The use of this smart contract is equivalent to the repartition key method currently used by WeSmart. Moreover, because Hyperledger Sawtooth is used for the implementation of our proposed solution, it is already possible for WeSmart to deploy smart contracts. The major advantages of using them rather than a repartion key being that the smart contract is immutable once written and accessible by all the members of the energy community.
However, this solution could not be directly applicable in Europe due to the regulation on data protection and privacy (GDPR) as explained in the section 3.4. Indeed, in this solution, the blocks of the blockchain contain information about the energy exchange of the members of the microgrid.
If in the future, solutions are found in order to record personal information in the

blockchain while respecting the GDPR rules, for example using anonymisation, then the solution described above could be a possible way of improving the current solution.

The current proposed solution performs a proof of data possession (see section 4). Indeed, our solution is able to quickly verify if the data stored in the cloud is intact. However, there is currently no mechanism that is able to restore the data from being modified (proof of retrievability). A solution that can be considered by WeSmart is to use error correcting code. As explained in the section 4.2, the central idea behind this concept is to encode the data with redundant informations before putting it in the cloud. The error correcting code permits to retrieve the original data from being affected by a limited number of modifications in the cloud.

An other limitation of the proposed solution is that the integrity verification of cloud data requires to download the entire data to compute the hash. In some situation, this can be too costly. A way to manage this issue is proposed in the article of Dongdong Y. et al. [20]. It consists in dividing data in several shards. The hash of every shards is computed and stored in the form of a Merkle tree in the cloud while the Merkle root is stored in the blockchain. In this way the blockchain ensures that the Merkle root can not be modified. The data integrity verification can then be performed by comparing a few shards and so by not downloading the entire data. There is a trade off between performance and reliability of the integrity check. At most shards are compared, at most the integrity check is reliable but it reduces the performances because more data need to be downloaded from the cloud. Conversely, comparing a low number of shards limits the data that need to be downloaded from the cloud but it increases the risk that a modification is performed in a shard that is not compared. A drawback of this method is that it requires additional space in the cloud to store the entire Merkle tree.

Finally, a hypothesis under which the solution was built is that the hash function used by WeSmart can not be altered (see section 5.1). If this assumption is not already satisfied, a solution could be to run the hash function in an SGX enclave. Moreover, if WeSmart initiates all its TLS connection in its enclave, then the value put on the blockchain by WeSmart would never leave a secure environment. Indeed, the information would be transferred from an enclave to the blockchain through a TLS connection or inversely.

8.2 Improvement of the blockchain concept

Common blockchains like the one used in our solution can not offer simultaneously a high throughput and a high scalability. This is because no consensus algorithm can guarantee high transaction throughput while maintaining a high number of nodes at the same time. For example, PBFT consensus provides a high throughput but is limited to a small number of nodes. Conversely, the PoW consensus works with a high number of node but has a low throughput. Article [32] proposes a solution for a blockchain to fulfill both high throughput and high scalability by using a hierarchical blockchain. Such a blockchain is composed of two tiered. The first tier is a Core Engine, which is based on a Practical Byzantine Fault Tolerance (PBFT) consensus to cope with a high throughput, that supervises the underlying subordinate engines (sub-engines) as its second tier. This low throughput second tier comprises different functionalities engines such as : the Payment, the Computation, and the Storage Engine. The consensus used for each sub-engine may vary and is chosen such that it is the most well-suited for a particular task. It can so be chosen in order to handle a large number of nodes.
The Core engine is a parent of all the other engines. Its role is to control the collaborations between the sub-engines. It provides feature such as sub-engines parallelism and interoperability between sub-engines. This tends to bring centralization to the architecture because the core engine becomes the supernodes and the ruler of all nodes. However it is mitigated by using a board of nodes that use the Practical Byzantine Fault Tolerant algorithm to govern this tier. This is a hybrid architecture with centralized and decentralized parts.

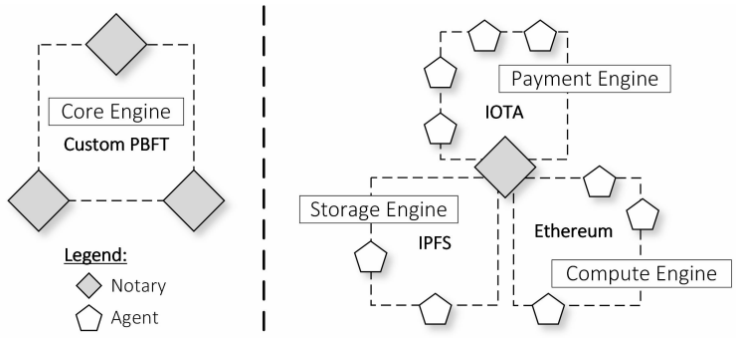


Figure 8.2.1: Multi-blockchain architecture proposed by Yustus Eko Oktian et. al. [32]

The article [26] presents an other implementation of a hierarchical blockchain that solves the problem of scalability and data access control. This scalability issue comes from the continuous increase of the blockchain size. For example, the size of

the Bitcoin blockchain was of 330GB in April 2021. This can lead to a centralized situation as less users are able to maintain such a large blockchain. Moreover the size increase can slowdown the blockchain performances. Futhermore, as most organizational structures are hierarchical, there is no solution in order to fulfill organisational needs. Private blockchains protect informations from outside but there is no specific mechanism to impose different data access control for different groups inside the same blockchain.

In response to these issues, the article of Swagatika Sahoo et. al. [27] proposes a hierarchical blockchain model where each level maintains multiple local blockchain-networks. This architecture facilitates the access control of blockchains differently to different groups of people or processes as per their access rights. This implementation is also more scalable because information can be distributed in the different tiers. Very precise information is stored in the deep tiers of the hierarchical blockchain while more general information is stored on the superficial tiers. The article gives an example where a policeman needs to verify if a user has a valid insurance. The objective when he queries the blockchain is to have a 'yes' or 'no' response. The policeman does not need to know all the detail of the insurance such as the insurance company name, the price... In this case, the policeman can ask a high level blockchain to know whether the user has a valid insurance while the details of this insurance are stored in a deeper level blockchain.

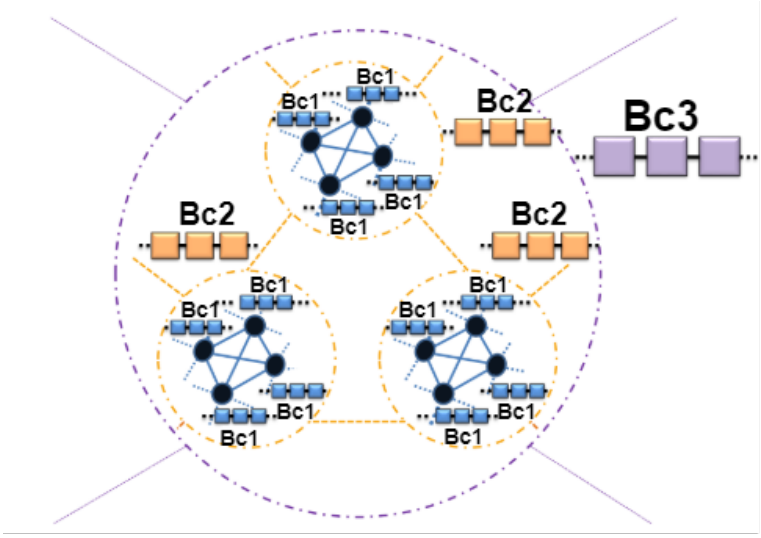


Figure 8.2.2: Structure of the hierarchical blockchain proposed in the article of Swagatika Sahoo et. al. [27]

Conclusion

This report first reviews the concept of the blockchain and the microgrids. The use of the blockchain in the energy sector has then been analysed. Although it has been shown that blockchains are able to ensure some essential features needed for the operation of the microgrid, implementation such as the Brooklyn microgrid that uses the blockchain as a trading platform can not already be implemented in Europe due to legal concerns.
This report also presents a solution to verify the cloud data integrity. This is performed with the use of a blockchain that stocks the hash of the information stored in the cloud data. The data integrity check is then performed by comparing the hash of the data stored in the cloud and the one saved in the blockchain.
This solution has been realized in partnership with the WeSmart company and is designed as an additional security layer on top of the existing WeSmart implementation.
The blockchain is implemented thanks to the Hyperledger Sawtooth tool which provides a high modularity to the solution. Indeed, blockchain settings such as authorizations or consensus can easily be modified. This allows to overpass limitation of non scalability of the PBFT consensus by switching to the PoET consensus if the number of participants in the blockchain becomes too high. Moreover additional functionalities can be developed by writing a new transaction family and running the transaction processor on top the existing blockchain. An interesting example of transaction family that can be used in the future is "Sabre" which allows to run smart contracts in an Hyperledger Sawtooth blockchain.
The last part of this work performs measurements on the implemented blockchain and highlights its protection mechanism against DoS attacks.

Part III

Annexes

Chapter 9

General concepts

The goal of this section is to clarify some prerequisite concepts useful for the understanding of this report.

9.1 Digital signature [47]

The digital signature uses the principle of public and private keys. Those keys are interesting tools to perform two different things : encryption and authentication. Encryption guarantees confidentiality and integrity. On the other hand, digital signatures guarantees authentication, non-repudiation and integrity.
Let's take an example in order to better understand those concepts. Imagine that Alice wants to send a private message to Bob. In this example, Alice owns 2 keys : her public key and her private key. Those keys are a series of bits. The public key can be known by everybody. On the other hands, only Alice knows her secret key. The situation is the same for Bob; everybody knows its public key but he is the only one to know its secret key.
The first thing that Alice wants to do is to encrypt the message that she wants to send to Bob so that only Bob can decrypt the message. In order to do that, she will encrypt its message with the public key of Bob. This message can only be decrypted thanks to the secret key of Bob. Because this secret key is only known by Bob, he is the only one to be able to decrypt the message sent by Alice. This guarantees confidentiality. Integrity is guaranteed by the fact that even a slight modification of the encrypted data will cause the decryption process to fail [94].

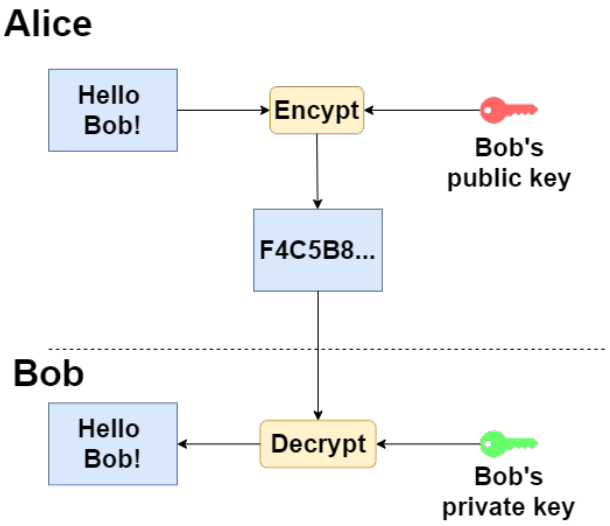


Figure 9.1.1: Encryption procedure

The second things that Alice may want to do is to sign her message so that Bob will be sure that Alice is the person who sent the message. To do that, Alice will combine her private key and the message to create a short signature bound to the message. Thanks to the message, the public key and the signature, Bob is able to verify that the message comes from the owner of the private who produces the message (Alice in this case). This ensures authentication. This verification also ensures that the message has not been tempered. Indeed, the process used to authenticate Alice needs the combination of the message, her public key and her digital signature. If the message was tampered, the digital signature would not correspond to the message anymore and the authentication process would reject Alice as the author of the message. This ensures the integrity of the message. Finally, non-repudiation is ensured by the fact that since the Alice is the only one with access to her private key used to apply the signature, she can not later claims that she was not the author of the message. It is also important to note that it is impossible to find the private key of Alice thanks to combination of the message, the signature and her public key. It is also impossible to find a valid signature corresponding to a message if the private key is not known.

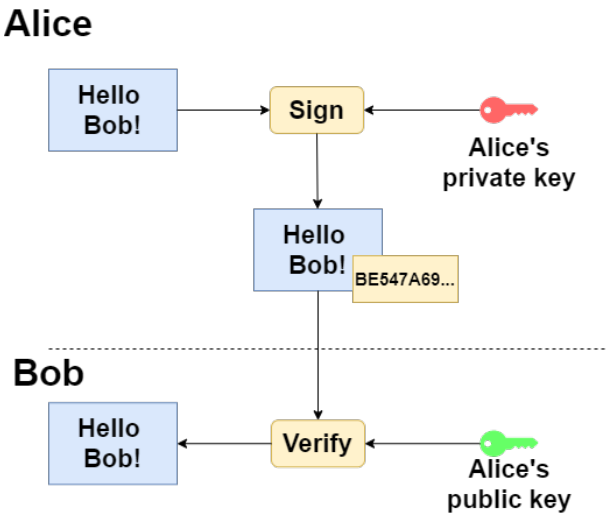


Figure 9.1.2: Signing procedure

Thanks to combination of those two principles, if an attacker is placed in the middle of the discussion and intercepts the messages, he will only be able to authenticates the writer of the message but he will not be able to read the message (since a private key is needed) neither tampered the message (otherwise, the receiver will observe that the message has been tampered).

9.2 Cryptographic hash function [93]

A cryptographic hash function is a function that takes a variable length file as input and converts it into a numerical values that looks random. Nonetheless, it only looks random. Indeed, hashing twice the same input will always give the same output. However, a small change on the input completely change the output result. Note that the output of any input has always the same length (defined by the type of hashing technology used). Because hash values are unique, they are also referred to as "fingerprints". Cryptographic hash functions have some properties :

- One-way: given a hash value, it is almost impossible to find the original value used to generate this hash value.
- Collision resistance: it is almost impossible to find two inputs that generate the same hash value.
- Random oracle property: a hash value is indistinguishable from a random n-bit value.

9.3 Failure [77]

The nodes of a distributed system may fail in four ways: crash-stop, omissions, crash-recovery, Byzantine.

- In the crash-stop case, a node stops to send and receive messages forever and does not recover.
- In the omission failure case, the node omits sending and/or receiving messages.
- In the crash-recovery failure case, the node stops receiving and/or sending messages. However, it may recover after crashing.
- In the Byzantine failure case, the node behaves arbitrarily and can send arbitrary messages to the other nodes. However, it can also behave maliciously and try to corrupt the system by sending false informations.

There is a hierarchy among the failure types. Indeed, as represented in the figure 9.3.1, the crash-stop is a particular case of the omission; it is the omission restricted to omitting everything after a certain event.
The omission is a special case of the crash recovery; assume a special case of crash-recovery where node uses only stable storage. In this case crash-recovery is identical to omission. Indeed, crashing, recovering and reading the last state from storage is equivalent to omitting send and receive while being crashed. However, if the node uses volatile memory then the recovered node might not be able to restore all of state. In this case, crash-recovery is an extension of omission with amnesia. Finally, crash-recovery is a special case of the Byzantine failure since Byzantine failures allow any behavior.

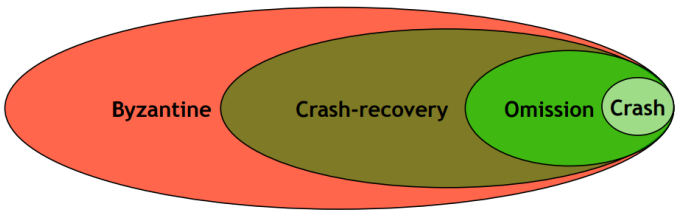


Figure 9.3.1: Failure Hierarchy [77]

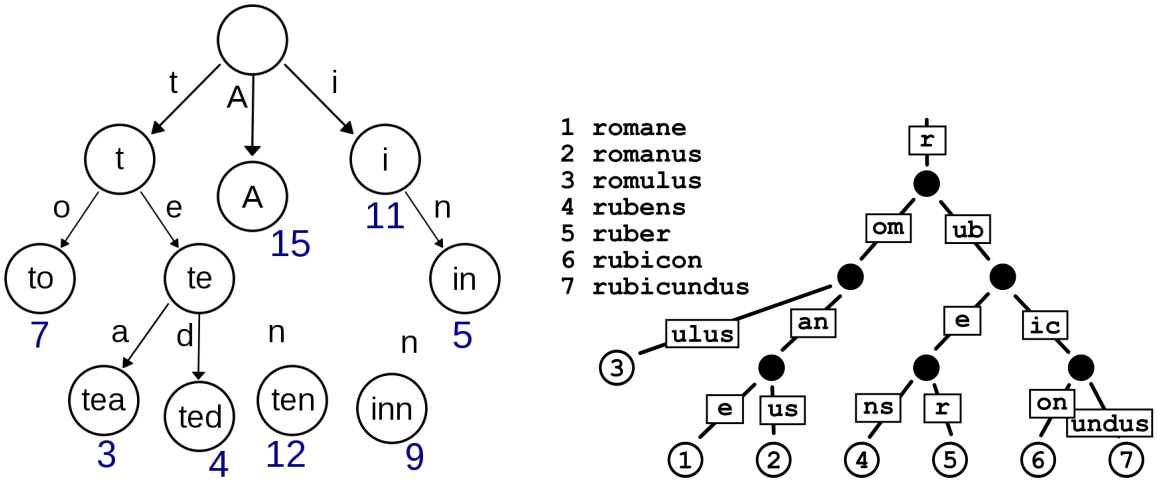
Chapter 10

Hyperledger Sawtooth

10.1 Prefix and Radix tree

A prefix tree is a data structure used for locating specific keys within a set. Most of the time, these keys are string. An example of prefix tree is given in figure 10.1.1.a. The link between two nodes does not represent the entire key but an individual character. The entire key is obtained by traversing the tree depth first following the links between nodes. In this example the value '3' is stored with the key "tea" and can be obtained by traversing the tree from root then following the edge 't', 'e' and 'a'.

A radix tree uses the same concept with the difference that the link between two nodes is a single character or a serie of characters. An example is given in the figure 10.1.1.b. This type of trees is more efficient for small sets and for sets of strings that share long prefixes.



a) Prefix tree [87] b) Radix tree [75]

Figure 10.1.1: Prefix and radix tree

10.2 Block, batch and transaction data structure

Transaction data structure

The transaction header contains the following fields :

- **batcher_public_key** : The public key of the client who added this transaction to a batch. This is because the transaction and the batch can be signed by different keys. For example a browser application signs the transaction and a server side adds transactions, creates and signs the batch. This functionality enables interesting deployment however it poses security problems. The fact that the transaction header contains the batcher public key prevents the transaction from being reused separately from the intended batch. It is not possible to take a batch and to repack it in a new batch unless you have the batcher private key.
- **dependencies** : A list of transaction signatures that describes the transactions that must be processed before this transaction can be valid.
- **family_name** : The family name correlated to the transaction processor's family name that this transaction can be processed on.
- **family_version** : The family version correlated to the transaction processor's family version that this transaction can be processed on.

- **inputs** : A list of addresses that are given to the context manager and controls what addresses the transaction processor is allowed to read from.
- **nonce** : A random string that ensures that two transactions can not be identical. In this way, even if all the other fields are the same, the header signature is guaranteed to be different.
- **outputs** : A list of addresses that are given to the context manager and controls what addresses the transaction processor is allowed to write to.
- **payload_sha512** : The sha512 hash function of the encoded payload.
- **signer_public_key** : The public key for the client that signed the transaction header.

The transaction message contains the following fields :

- **header** : The serialized version of the transaction header.
- **header_signature** : The signature derived from signing the header.
- **payload** : The encoded family specification of the transaction. It is used by the transaction processor to perform changes in the state.

Batch data structure

The batch header is composed of the following fields :

- **signer_public_key** : The public key of the client that signed the batch header.
- **transaction_ids** : List of transaction.header_signatures that matches the order of transactions required for the batch.

The batch message is composed of the following fields :

- **header** : The serialized version of the batch header.
- **header_signature** : The signature derived from signing the header.
- **transactions** : A list of the transactions that matches the list of transaction_ids listed in the batch header.
- **trace** : A debugging flag that indicates that this batch should be traced through the system, resulting in a higher level of debugging output.

Block data structure

The block header is composed of the following fields :

- **block_num** : The block number in the chain.
- **previous_block_id** : The header signature of the previous block in the chain.
- **signer_public_key** : The public key of the validator that signed the block header.
- **batch_ids** : The list of batch.header_signature that matches the order of batches required for the block.
- **consensus** : Bytes that are set and verified by the consensus algorithm used to create and validate the block.
- **state_root_hash** : The state_root_hash should match the final state_root after all transactions in the batches have been applied, otherwise the block is not valid.

The block message is composed of the following fields :

- **header** : The serialized version of the block header.
- **header_signature** : The signature derived from signing the header.
- **batches** : The list of batches contained in the block.

10.3 Journal

Journal overview

The journal is the group of validator subcomponents that works together to handle batches and propose blocks. These components are responsible for completing published blocks, publishing batches into blocks to extend the chain and validating proposed blocks to determine if they should be considered for the new chain head.

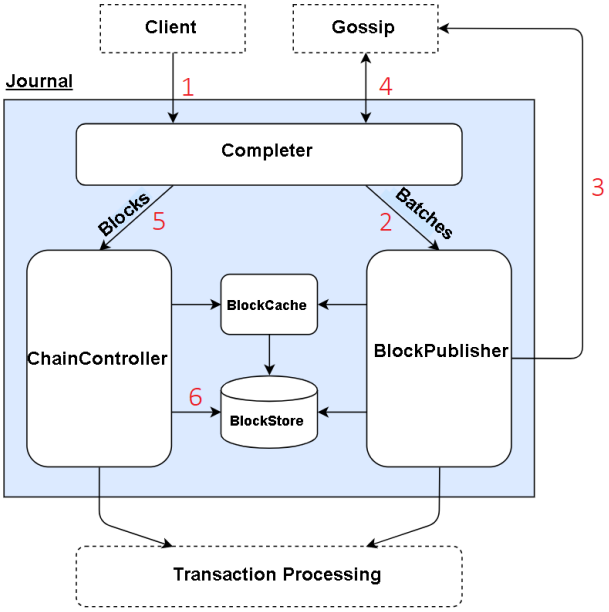


Figure 10.3.1: Architecture of the journal

The architecture of the journal is shown in the figure 10.3.1. Blocks and batches arrive via interconnect, either through the gossip protocol or from client requests. Then the processing of these blocks and batches is handled in multiple pipelines :

- The Completer initially receives the blocks and batches. It guarantees that all dependencies for the blocks and batches have been satisfied.
- Completed batches go to the BlockPublisher for batch validation and inclusion in a block.
- Completed blocks go to the ChainController for block validation and fork resolution.
- The BlockCache and BlockStore provide storage for the batches and blocks being processed.

The advantage of this structure is that batch and block processing can be performed asynchronously. The ChainControler and the BlockPublisher can work together in parallel with different block/batch rate. Moreover this structure is flexible enough to work with different consensus algorithms.

The completer

The completer is responsible for making sure that blocks and batches are complete before delivering them to the BlockPublisher or ChainController. When a proposed block is broadcasted, it contains only the minimum required information, such as batch IDs but not the batch themselves. The completer checks for dependencies : it ensures that the previous block exists and makes sure that batches exist in the BlockStore or BlockCache.

- A batch is considered complete when all of its dependent transactions exist in the current chain or have been delivered to the BlockPublisher. Once a batch is formally complete, it is delivered to the BlockPublisher.
- A block is considered complete when all of its predecessors have been delivered to the ChainController and the batches field contains all the batches specified (as batch IDs) in the block header. Also, the batches field must be in the same order as the list of batch IDs. Once a block is formally complete, it is delivered to the ChainController for validation.

Note that all blocks and batches have a timeout for being completed. The Completer sends an initial request for any missing dependencies or predecessors. If a response is not received in the specified time, the block or batch is dropped.

The BlockPublisher

The BlockPubisher is responsible for creating candidate blocks to extend the current chain. The BlockPublisher does all of the housekeeping work for creating a block, but takes direction from the consensus algorithm for when to create a block and when to publish a block.

The BlockPublisher is responsible for creating new candidate blocks. As batches are received by the validator (from clients or other Sawtooth nodes), they are added to the BlockPublisher's pending queue. Only valid transactions will be added to the next candidate block. For timeliness, batches are added to a scheduler as they are added to the pending queue; thus, transactions are processed incrementally as they are received.

When the pending queue changes significantly, such as when the chain head has been updated by the Chain Controller, the Block Publisher cancels the current scheduler and creates a new scheduler.

The ChainController

The ChainController is responsible for maintaining the blockchain for the validator. This responsibility involves validating proposed blocks, evaluating valid

blocks to determine if they should be considered for the new chain head, and generating new blocks to extend the chain. The ChainController determines which chain the validator is currently on and coordinates any change-of-chain activities that need to happen.

The ChainController maintains the current chain head (a pointer to the last block in the current chain). When it receives a candidate block (over the network or from the Block Publisher), it determines whether the chain head should be updated to point to that candidate block. The Chain Controller creates a scheduler to calculate new state with a related Merkle hash for the block being published. The Merkle hash is compared to the state root contained in the block header. If they match, the block is valid from a transaction execution and state standpoint. The Chain Controller uses this information in combination with consensus information to determine whether to update the current chain head.

The BlockStore

The block store is a persistant on-disk store of all blocks in the current chain. It is the list of blocks from the current chain head back to the genesis block. All the blocks stored in the BlockStore are formally complete. In addition, the BlockStore maintains internal mappings of transaction-to-block and batch-to-block. These mappings are stored in a format that is cached to disk, so they are not held in memory at all times. Note that as the blockchain grows, this set of mappings will become quite large.

Blocks in the BlockStore can be accessed by block ID. Blocks can also be accessed via batch ID, transaction ID, or block number.

The BlockCache

The BlockCache is an in-memory construct that is rebuilt when the system is started. It holds the working set of blocks for the validator.

The BlockCache tracks the processing state of each block as valid, invalid, or unknown.

- Valid blocks have been proven to be valid by the ChainController. All blocks in the BlockStore are considered valid.
- Invalid blocks have failed validation or have an invalid block as a predecessor.
- Unknown blocks have not yet completed validation. Usually, these blocks have just arrived from the Completer.

If a block is not present in the BlockCache, the validator looks in the BlockStore for the block. If it is not found or the lookup fails, the block is marked as unknown.

If the block is found in the BlockStore, it is loaded into the BlockCache and marked as valid.

The BlockCache keeps blocks that are currently relevant, tracked by the last time the block was accessed. Periodically, blocks that have not been accessed recently are purged from the BlockCache, but only if none of the other blocks in the BlockCache reference those blocks as predecessors.

The consensus interface

In the spirit of configurability, Sawtooth supports dynamic consensus. The initial consensus algorithm for the blockchain is set in the genesis block, but can be changed during a blockchain's lifetime with the Settings transaction processor.

The consensus interface is responsible for determining who can publish a block, whether a published block is valid according to the consensus rules, and which block should become the chain head in the case of a fork.

10.4 Handshake procedure

10.4.1 Trust authorization

In a "trust authorization" case, the message diagram is represented in the figure 6.2.6.a. This is the simplest authorization type. The requester sends an authorizationTrustRequest message with the role it wants to access and the public key of the requester. The requester can make request the role ALL to have access to all the available roles. The validator trusts the public key in the AuthorizationTrustRequest and approves any roles requested that are available on that endpoint. However if a role requested is not available, the requester is rejected and the connection is closed.

```
message ConnectionRequest {
  // This is the first message that must be sent to start off authorization.
  // The endpoint of the connection.
  string endpoint = 1;
}

enum RoleType {
  // A shorthand request for asking for all allowed roles.
  ALL = 0;

  // Role defining validator to validator communication
  NETWORK = 1;
}

message ConnectionResponse {
  // Whether the connection can participate in authorization
  enum Status {
    OK = 0;
    ERROR = 1;
  }

  //Authorization Type required for the authorization procedure
  enum AuthorizationType {
    TRUST = 0;
    CHALLENGE = 1;
  }

  message RoleEntry {
    // The role type for this role entry
    RoleType role = 1;

    // The Authorization Type required for the above role
    AuthorizationType auth_type = 2;
  }

  repeated RoleEntry roles = 1;
  Status status = 2;
}
```

Figure 10.4.1

```
message AuthorizationTrustRequest {
  // A set of requested RoleTypes
  repeated RoleType roles = 1;
  string public_key = 2;
}

message AuthorizationTrustResponse {
  // The actual set the requester has access to
  repeated RoleType roles = 1;
}
```

Figure 10.4.2

10.4.2 Challenge authorization

In a "challenge authorization" case, the message diagram is represented in the figure 6.2.6.b. The requester sends first an AuthorizationChallengeRequest to request a challenge to sign. The validator responds with an AuthorizationChallengeResponse message that contains a random payload that the requester must sign. The requester signs this payload and responds with an AuthorizationChallengeSubmit message. It contains the public key of the requester, the signature derived from signing the challenge payload and the requested role. Note that the requester can also request ALL. The validator responds with an AuthorizationChallengeResult which says if the challenge is accepted. Contrary to the authorization trust, the validator checks the public key of the requester thanks to the challenge. If the requester tries to cheat with a false public key, it does not know the private key associated with this public key and is not able to sign the challenge. If the public key is verified against the signature and the public key is included in the policy, the node response is accepted. The requester can then start sending messages for the approved roles. However if a role requested is not available, the requester is rejected and the connection is closed.

```
message AuthorizationChallengeRequest {
  // Empty message sent to request a payload to sign
}
```

Figure 10.4.3

```
message AuthorizationChallengeResponse {
  // Random payload that the connecting node must sign
  bytes payload = 1;
}
```

Figure 10.4.4

```
message AuthorizationChallengeSubmit {
  // public key of node
  string public_key = 1;

  // signature derived from signing the challenge payload
  string signature = 3;

  // A set of requested Roles
  repeated RoleType roles = 4;
}
```

Figure 10.4.5

```
message AuthorizationChallengeResult {
  // The approved roles for that connection
  repeated RoleType roles = 1;
}
```

Figure 10.4.6

Note that if the requester deviates from the procedure described above in any way, the requester is rejected and the connection is closed. The same is true if the requester sends multiple connectionRequest messages or a multiple of any authorization-type message. If too many messages or if they are too close together, the connection may be considered malicious and is rejected.

10.5 Authorization violation

If at any time a requester tries to send a message that is against its allowed permission, the validator responds with an AuthorizationViolation message and the connection is closed. If that requester wishes to rejoin the network, it will need to go back through the connection and authorization process described above.

```
message AuthorizationViolation {
  // The Role the requester did not have access to
  RoleType violation = 1;
}
```

Figure 10.5.1

Chapter 11

Deployment of the solution

In this part, the deployment of the blockchain on multiple nodes is explained and the importance of the NAT is clarified.
This document is based on the Hyperledger Sawtooth documentation [89]. Note that Ubuntu 18.04 (Bionic) is required on every node and that at least 4 nodes are required in the network to perform the consensus.

11.1 Installation of Sawtooth on all the nodes

The first step is to install Hyperledger Sawtooth on all the nodes. This can be done with the following commands :

```
\$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
--recv-keys 8 AA7AF1F1091A5FD
\$ sudo add-apt-repository 'deb [arch=amd64]
http://repo.sawtooth.me/ubuntu/chime/stable bionic universe'
```

Update the package list :

```
\$ sudo apt-get update
```

Install the Sawtooth package

```
\$ sudo apt-get install -y sawtooth
```

Install the PBFT consensus engine package

```
\$  sudo apt-get install -y sawtooth sawtooth-pbft-engine
```

At this moment, some files have to be added for our implementation to work properly. The files "sawtooth_we" and "sawtooth_we-1.2.3.egg-info" need to be added in usr/lib/python3/dist-packages. The files "we" and "we-tp-python" need to be added in usr/bin.
The files "sawtooth_we", "sawtooth_we-1.2.3.egg-info", "we" and "we-tp-python" are contained in the folder that we provided to you.

11.2 Creation of user and validator keys

Generate the user keys :

```
\$ sawtooth keygen my_key
writing file: /home/yourname/.sawtooth/keys/my_key.priv
writing file: /home/yourname/.sawtooth/keys/my_key.pub
```

Generate the keys for the validator :

```
\$ sudo sawadm keygen
writing file: /etc/sawtooth/keys/validator.priv
writing file: /etc/sawtooth/keys/validator.pub
```

The user keys (also called transaction keys) are used to sign batches and transactions that a user wants to submit to the validator. Thanks to them, it is possible to control which users can submit transactions and batches to a validator (see section 11.10).
On the other hand the validator keys are used to sign the blocks that the validator produces. Those keys are also interesting in a network point of view. Indeed, it is the validator that is connected to the other nodes. Thanks to the validator keys, it is possible to control the identity of a nodes before it accesses an existing network blockchain (see section 11.10).

11.3 Creation of the genesis block

The genesis block is the first block of the blockchain. It specifies the initial on-chain settings for the network configuration. This step should be executed only on one node of the blockchain. The other nodes will recover the genesis block when they join the network.

Change to a writable directory :

```
\$ cd /tmp
```

Then create a batch with a settings proposal for the genesis block. This command authorizes the node that creates the genesis block to set and change Sawtooth settings :

```
\$ sawset genesis --key $HOME/.sawtooth/keys/my_key.priv \
-o config-genesis.batch
```

Create a bacth to initialise the PBFT consensus settings.

```
\$ sawset proposal create --key
$HOME/.sawtooth/keys/my_key.priv \
-o config-consensus.batch \
sawtooth.consensus.algorithm.name=pbft \
sawtooth.consensus.algorithm.version=1.0 \
sawtooth.consensus.pbft.members=
'["VAL1KEY","VAL2KEY",...,"VALnKEY"]'
```

On this command, "VAL1KEY", "VAL2KEY", "VAL3KEY", ..., "VALnKEY" are the validator public keys of all the nodes of the network (including the node that creates the genesis). This public key can be found in the file /etc/sawtooth/keys/-validator.pub on each node. Be sure to use single quotes and double quotes correctly, as shown in the example of figure 11.3.1.

Combine the separate batches into a single genesis batch that will be committed in the genesis block :

```
\$ sudo -u sawtooth sawadm genesis \
config-genesis.batch config-consensus.batch
```

The figure 11.3.1 shows the creation of the genesis block.

11.4 Configure Peers in Off-Chain Settings

For the PBFT consensus, each node specifies the peer nodes in the network, because a PBFT network must be fully peered (all nodes must be directly connected). This is an off-chain setting that is set in the **validator.toml** file.

Create the **validator.toml** file by copying the example file :

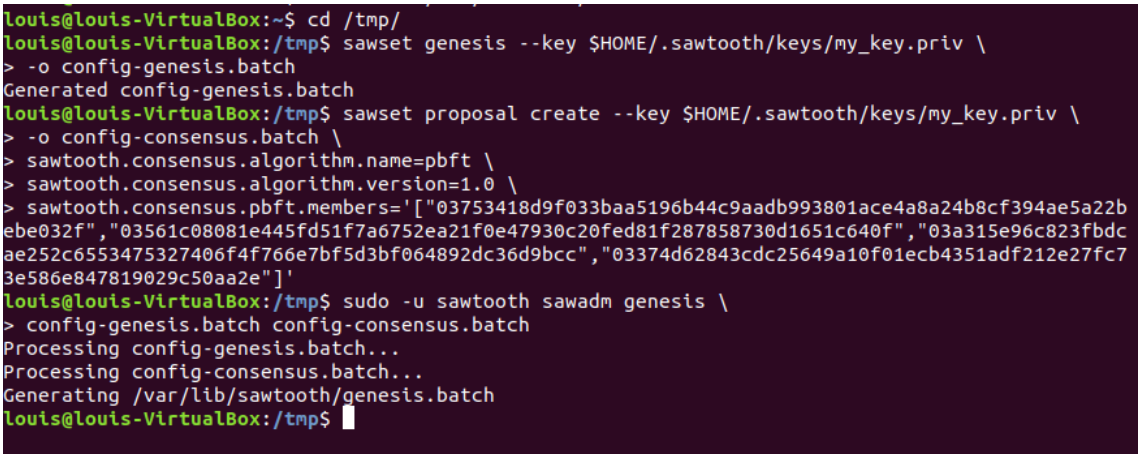


Figure 11.3.1: Creation of the genesis block

```
\$ sudo cp -a /etc/sawtooth/validator.toml.example
/etc/sawtooth/validator.toml
```

Edit this file :

```
\$ sudo nano /etc/sawtooth/validator.toml
```

There are different settings on this file:

- The peering setting should be set to 'static'
- The bind-network setting specifies where the validator listens for communication from other nodes. It can be set to tcp://enp0s3:8800. Note that the port number has to be the same that the port number of the endpoint setting.
- The bind-component setting specifies where the validator listens for communication from this validator's components, such as the REST API and transaction processors. It can be set to tcp://127.0.0.1:4004
- The bind-consensus setting specifies where the validator listens for communication from the consensus engine. It can be set to tcp://127.0.0.1:5050
- The endpoint setting specifies the address that other peers should use to find the validator on this node. You will also specify this value in the peer list when starting a validator on another node. If the blockchain is deployed in a local network (all the nodes are on the same wifi), this endpoint can be

set with the private IP address of the node on port 8800. If the blockchain is deployed on a global network, endpoint should be set with the publicly addressable IP address and port. A more detailed example based on a NAT system is described in the section 11.14.

- The peer setting specifies the addresses that this validator should use to connect to the other nodes (peers); that is, the public endpoint strings of those nodes. As for the endpoint setting, it can be set with the private IP address or the publicly addressable IP address depending on the type of network.

- The network_public_key and network_private_key : These two keys should be the same for all the nodes. They are used to secure the communication between nodes. (See section 11.6 for generating them properly).

- The minimum_peer_connectivity : Set the minimum number of peers required before stopping peer search.

- The maximum_peer_connectivity : Set the maximum number of peers that will be accepted.

The figure 11.4.1 shows the validator.toml file. Note that in this example, the blockchain is created in a local network and so the endpoint and peers are set with the private IP address of the nodes.
The figure 11.4.2 shows an other example where the blockchain is deployed on a global network. In this configuration, the endpoint and peers are set with the public IP addresses and a port mapping is performed in the router (cf. section 11.14). It is interesting to note that the port used for the bind-network is the same than the port for the endpoint.

```
# The following is a possible example.

# Bind is used to set the network and component endpoints. It should be a list
# of strings in the format "option:endpoint", where the options are currently
# network and component.
bind = [
  "network:tcp://enp0s3:8800",
  "component:tcp://127.0.0.1:4004",
  "consensus:tcp://127.0.0.1:5050"
]

# The type of peering approach the validator should take. Choices are 'static'
# which only attempts to peer with candidates provided with the peers option,
# and 'dynamic' which will do topology buildouts. If 'dynamic' is provided,
# any static peers will be processed first, prior to the topology buildout
# starting.
peering = "static"

# Advertised network endpoint URL.
endpoint = "tcp://192.168.0.52:8800"

# Uri(s) to connect to in order to initially connect to the validator network,
# in the format tcp://hostname:port. This is not needed in static peering mode
# and defaults to None. Replace host1 with the seed's hostname or IP address.
# seeds = ["tcp://host1:8800"]

# A list of peers to attempt to connect to in the format tcp://hostname:port.
# It defaults to None. Replace host1 with the peer's hostname or IP address.
peers = ["tcp://192.168.0.21:8800", "tcp://192.168.0.43:8800", "tcp://192.168.0.53:8800"]

# The type of scheduler to use. The choices are 'serial' or 'parallel'.
scheduler = 'parallel'

# A Curve ZMQ key pair are used to create a secured network based on side-band
# sharing of a single network key pair to all participating nodes.
# Note if the config file does not exist or these are not set, the network
# will default to being insecure.
#network_public_key = 'wFMwoOt>yFqI/ek.G[tfMMILHWw#vXB[Sv}>l>i)'
#network_private_key = 'r&oJ5aQDj4+V]p2:Lz70Eu0x#m%IwzBdP(}&hWM*'
network_public_key = 'KDps$]PVP@^tG^Oph%u}&cigUnLip%4&>@vkc>+/'
network_private_key = 'SvV?h0ye(Rgx7gfHB>N23t#yr}7u4G{4JPhp(7LQ'
```

Figure 11.4.1: validator.toml local network

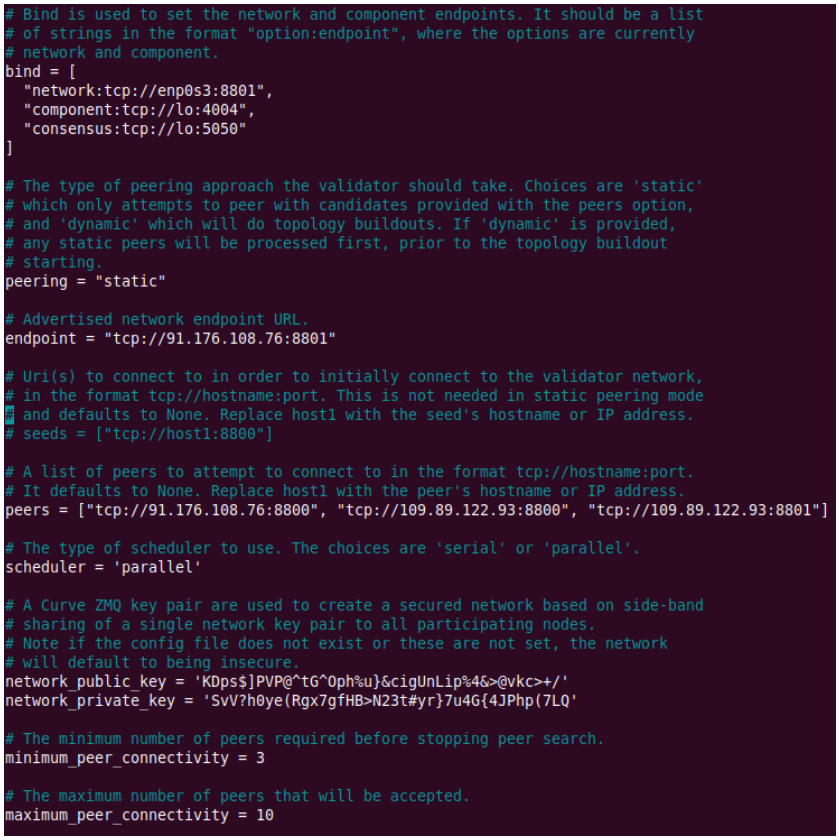


Figure 11.4.2: validator.toml global network

11.5 About bind-network setting

For the network bind string, you would typically use a specific network interface that you want to bind to. The **ifconfig** command provides an easy way to determine what this interface should be. **ifconfig** displays the network interfaces on your host system, along with additional information about the interfaces. For example:

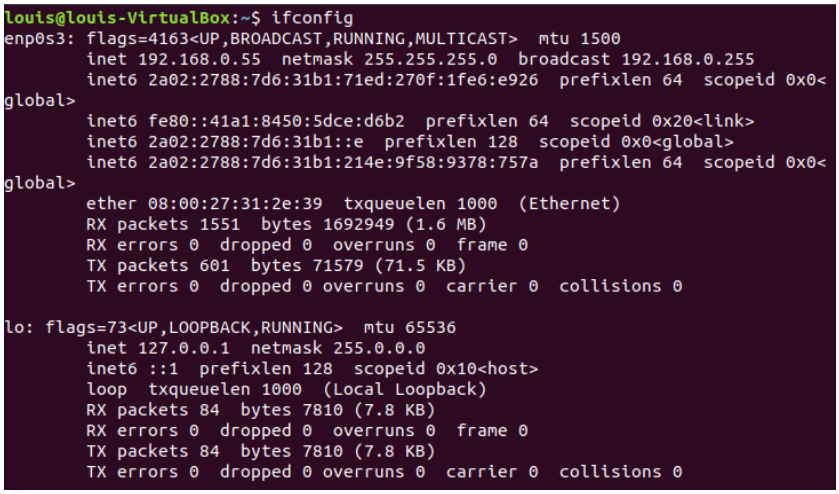


Figure 11.5.1: Result of the **ifconfig** command

This example output shows that **enp0s3** is a network interface that has access to the Internet. In this case, if you would like the validator to accept connections from other nodes on the network behind **enp0s3**, you could specify a network bind string such as **tcp://enp0s3:8800**.

11.6 Generating network_public_key and network_private_key

In order to secure the communication between the nodes in the network, the "network_public_key" and "network_private_key" can be changed.
The figure 11.6.1 shows how to use Python to generate those keys:

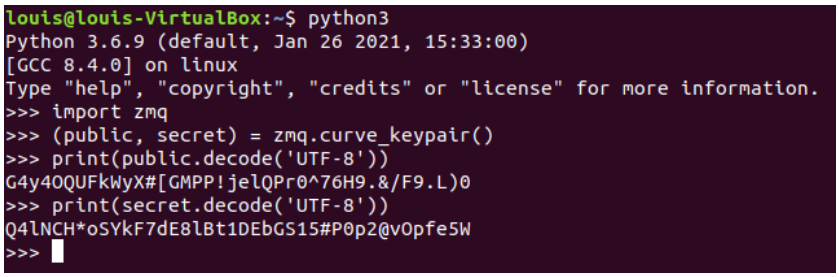


Figure 11.6.1: Generation of new network public and private key

Those values must replace the old values of "network_public_key " and "network_private_key " on all the nodes of the network.

11.7 Start Sawtooth on the first node

Now you can start the validator with the same settings than for the previous section :

```
\$ sudo -u sawtooth sawtooth-validator \
--bind component:{component-bind-string} \
--bind network:{network-bind-string} \
--bind consensus:{consensus-bind-string} \
--endpoint {public-endpoint-string} \
--peers {peer-list}
```

```
louis@louis-VirtualBox:/tmp$ sudo -u sawtooth sawtooth-validator \
> --bind component:tcp://127.0.0.1:4004 \
> --bind network:tcp://enp0s3:8800 \
> --bind consensus:tcp://127.0.0.1:5050 \
> --endpoint tcp://192.168.0.52:8800 \
> --peers tcp://192.168.0.21:8800,tcp://192.168.0.43:8800,tcp://192.168.0.53:8800
[2021-03-09 15:37:31.239 WARNING  (unknown file)] [src/pylogger.rs: 40] Started logger at level
WARN
```

Figure 11.7.1: Creation of the validator

On a separate terminal start the REST-API :

```
\$ sudo -u sawtooth sawtooth-rest-api -v
```

```
louis@louis-VirtualBox:~$  sudo -u sawtooth sawtooth-rest-api -v
[sudo] password for louis:
[2021-03-09 15:38:37.131 INFO     messaging] Connecting to tcp://localhost:4004
[2021-03-09 15:38:37.133 INFO     rest_api] Creating handlers for validator at t
cp://localhost:4004
[2021-03-09 15:38:37.142 INFO     rest_api] Starting REST API on 127.0.0.1:8008
======== Running on http://127.0.0.1:8008 ========
(Press CTRL+C to quit)
```

Figure 11.7.2: Creation of the rest-API

On a separate terminal start the setting transaction processor :

```
\$ sudo -u sawtooth settings-tp -v
```

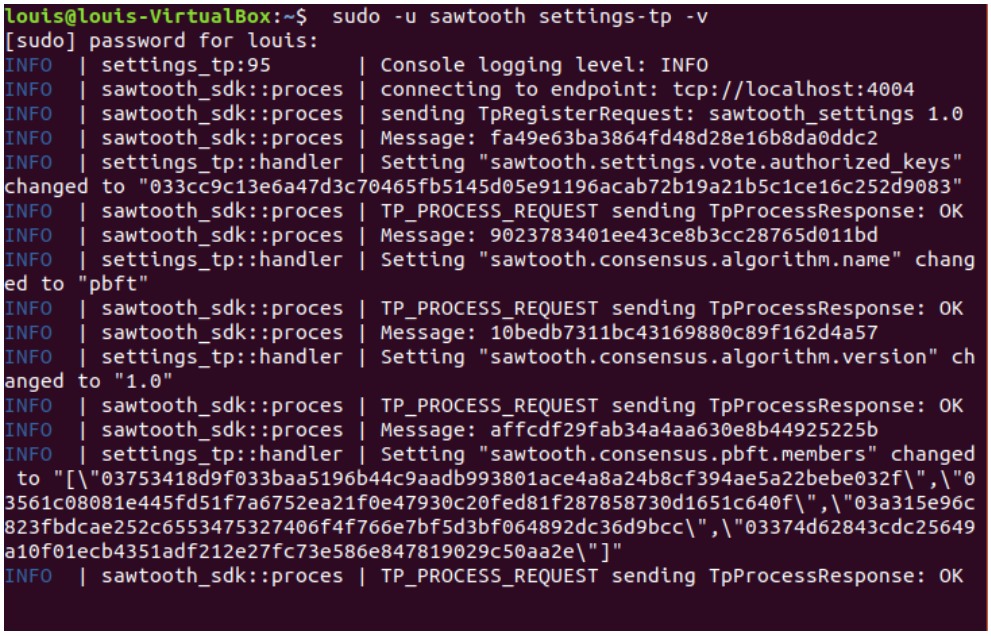


Figure 11.7.3: Creation of the setting tp

The Settings transaction family provides a methodology for storing on-chain configuration settings.

On a separate terminal start the identity transaction processor :

```
\$ sudo -u sawtooth identity-tp -v
```

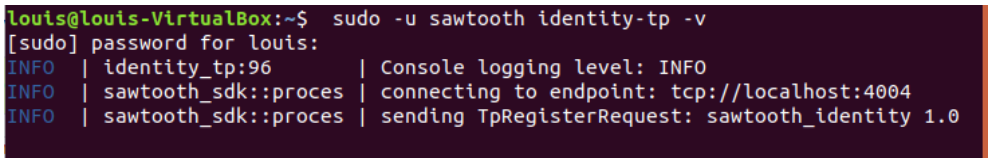


Figure 11.7.4: Creation of the identity tp

The identity transaction processor will be useful when the roles and policies will be created (see section 11.10)

On a separate terminal start the WeSmart transaction processor :

```
\$ sudo -u sawtooth we-tp-python -v
```

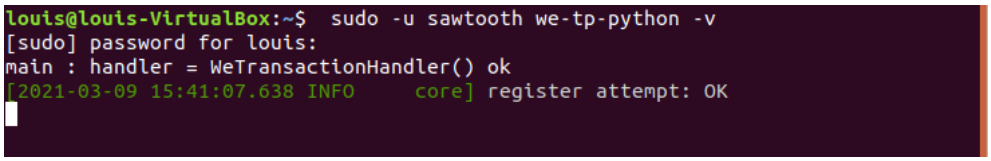


Figure 11.7.5: Creation of the We tp

On a separate terminal start the PBFT consensus engine :

```
\$ sudo -u sawtooth pbft-engine -vv --connect
tcp://localhost:5050
```

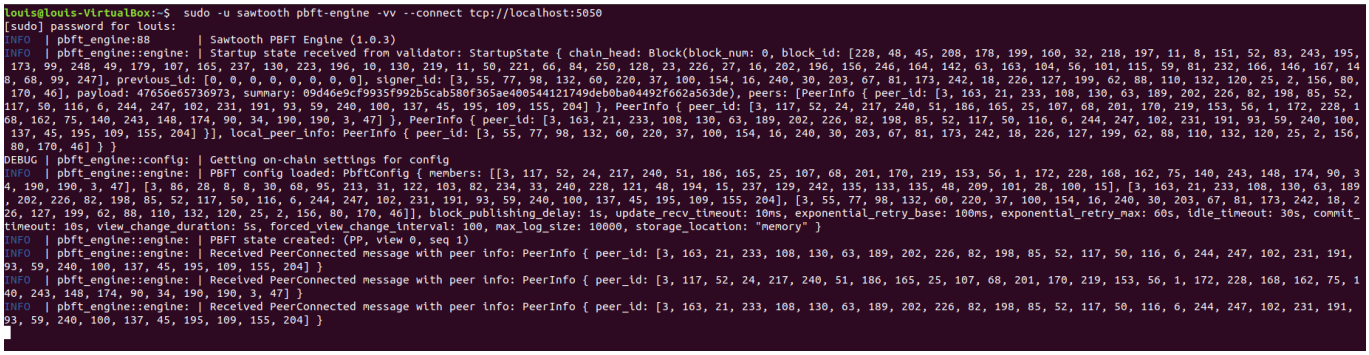


Figure 11.7.6: Creation of the PBFT engine

11.8 Start the other nodes

Once the first node created the genesis block, the other nodes can join the network. Note that at least 4 nodes are required on the network in order to perform the PBFT consensus. To join the network, each node has to set its **validator.toml** file as explained in section 11.4 and start the different terminals as explained in section 11.7.

11.9 Confirm network functionality

The commands of this section can be used on all the participating nodes of the blockchain.

Display the block contained on the blockchain with the following command. Note that at the beginning there is only the genesis block in the blockchain.

```
\$ sawtooth block list
```

The next command displays the peers with which the node is connected. Verify that each node is connected to all the other nodes. Note that the same IP address can appear multiple times.

```
\$ sawtooth peer list
```

The next step is to add a block in the blockchain. This is performed with the **set** command. Here is an explanation of the three parameters needed for this command.

- -date corresponds to the date and hour related to the energy consumptions. Thanks to this parameter, it is possible to recover the right consumption when a **get** is performed.
- -listId is the hash of the list of the ID of the participants of the renewable energy community.
- -listConsumption is the hash of the list of the consumption corresponding to the listId.

Note that if two **set** are performed with the same -date, the last **set** will overwrite the previous one.

```
\$ we set -date ... -listId ... -listConsumption ...
```

Recover data from the blockchain. The **get** command returns the listId and the listConsumption corresponding to a -date :

```
\$ we get -date
```

The figure 11.9.1 shows a scenario where a block is added in the blockchain. The first line shows the list of the peers to which the node is connected. In this case, the node is connected to 3 other nodes. There are 4 nodes in the blockchain and the PBFT consensus can be performed. The next command (**sawtooth block list**) shows that there is only the genesis block in the blockchain. Then a block is added to the blockchain with the **set** command. The data can be recovered with a **get** command with the same date as parameter. Note that the second **sawtooth block list** shows that a block has been added on the blockchain by the **set** command.

```
colette@colette-VirtualBox:~$ sawtooth peer list
tcp://192.168.0.42:8800,tcp://192.168.0.43:8800,tcp://192.168.0.52:8800
colette@colette-VirtualBox:~$ sawtooth block list
NUM  BLOCK_ID
                    BATS  TXNS  SIGNER
0    e4302dd0b2c7a020dac50b08973453f3c3ad63f831b36ba5ed82dfc40a82db0b32dd4254fa8017e21b10cac49cf6a48e3fa3683865733b
51e8a692a7944463f7  2     4     03374d...
colette@colette-VirtualBox:~$ we set -name 1200 -listId 1 2 3 -listConsumption 4 5 6
Response: {
  "link": "http://127.0.0.1:8008/batch_statuses?id=5c847df50741cc0e43c3d0390a65f3626cf4e6f58ab1e725c58cf18dbc7f5bb7
490a60f7fc0f684cc53f989a694f42e3de0170765ac15c02c60ab1ea2fc1a385"
}
colette@colette-VirtualBox:~$ we get -name 1200
Response :  1200-1,2,3-4,5,6
colette@colette-VirtualBox:~$ sawtooth block list
NUM  BLOCK_ID
                    BATS  TXNS  SIGNER
1    48a90e6060cc4f9e95bde4263e3a4d57ca949a0b44997e1e4bbec8b47cde60a00522dba45a5f485f23e58d6076716631ab60118762b91b
5ba8d5bac32cb0343c  1     1     03a315...
0    e4302dd0b2c7a020dac50b08973453f3c3ad63f831b36ba5ed82dfc40a82db0b32dd4254fa8017e21b10cac49cf6a48e3fa3683865733b
51e8a692a7944463f7  2     4     03374d...
```

Figure 11.9.1: Add a block in the blockchain

11.10 Limit the blockchain access

This section adds settings to the existing blockchain in order to make it private. There are two types of permissioning to limit the access to the blockchain :

- The transactor key permissioning controls who (which users and clients) can submit transactions and batches to a validator. These settings can be configured both on-chain and off-chain. In this implementation it is done in an on-chain manner.
- The validator key permissioning controls which nodes can connect to the Sawtooth network. These settings are configured with on-chain settings.

11.11 Transactor key permissioning

The permissioning is defined with a pair role-policy. The role defines the authorizations and the policy defines the list of nodes associated with this role.

The permissioning should be done by the node who created the genesis block. The first step is to add its public key to the list of those allowed to change settings:

```
\$ sudo sawset proposal create --key
$HOME/.sawtooth/keys/my_key.priv \
```

Tap "Enter" and then :

```
sawtooth.identity.allowed_keys=
$(cat ~/.sawtooth/keys/my_key.pub)
```

In order to perform the next command, you first have to create a copy of my_key.pub and my_key.priv respectively yourname.pub and yourname.priv. After that, you will be able to create a policy with the public keys of all the nodes allowed to submit transactions and batches. These keys are in the file **/home/yourname/.sawtooth/keys/my_key.pub**. Note that in the following command every node can commit a block.

```
\$ sawtooth identity policy create policy_batch "PERMIT_KEY *"
```

Then, you have to create a role for this policy. The transactor role is the top level role for controlling who can sign transactions and batches on the system. This role is used when the allowed transactors for transactions and batches are the same. Any transactor whose public key is in the policy is allowed to sign transactions and batches, unless a more specific sub-role disallows their public key.

```
\$ sawtooth identity role create transactor policy_batch
```

It is possible to display the policy and the role settings with the commands :

```
\$ sawtooth identity policy list
\$ sawtooth identity role list
```

The figure 11.11.1 shows the creation of a transactor policy. The transactor role is then set to this policy. Note that given it is an on chain setting, blocks are created to store the policy. In this example, the blocks 2, 3 and 4 are created when the policy and the role are committed.

The figure 11.11.2 shows that a new node that is not in the transactor policy can have access to the network however it can not commit a new block.

```
louis@louis-VirtualBox:~$ sawtooth block list
NUM  BLOCK_ID                                                      BATS  TXNS  SIGNER
1    3fd6218652d7825d68d1ea98160b7936d28fa0a7758d3d1c0eeae9f04fc950742fe1d080f7e
2be584782f079399f0df2bc1568cdd368e650fd17bebc311e15be  1     1     03561c...
0    499e7bac4815c66cfb636224d44dd487db1e6c2ec4023404331a298e2de385ac150b5102f61
1d19f1f64c603d9b3aaaa307a5366f3d9548194195f0afa480616  2     4     03374d...
louis@louis-VirtualBox:~$ sudo sawset proposal create --key $HOME/.sawtooth/keys
/my_key.priv \
>   sawtooth.identity.allowed_keys=$(cat ~/.sawtooth/keys/my_key.pub)
[sudo] password for louis:
louis@louis-VirtualBox:~$ sawtooth identity policy create policy_batch "PERMIT_K
EY 033cc9c13e6a47d3c70465fb5145d05e91196acab72b19a21b5c1ce16c252d9083" "PERMIT_K
EY 021f464aef19c2b076dc859f252e3790d7728ba90ae4f077fff3ec6c88b8f1f80f" "PERMIT_K
EY 0224e12a5b349047253fb706ce6741b17307603fbc86cb13d6b02f4defbe9d1ec5" "PERMIT_K
EY 02954e5b20976d4a308e003917740e01e18893275a462f67993693610a5afc25e7"
Policy committed in 1.20652 sec
louis@louis-VirtualBox:~$ sawtooth identity policy list
policy_batch:
  Entries:
    PERMIT_KEY 033cc9c13e6a47d3c70465fb5145d05e91196acab72b19a21b...
    PERMIT_KEY 021f464aef19c2b076dc859f252e3790d7728ba90ae4f077ff...
    PERMIT_KEY 0224e12a5b349047253fb706ce6741b17307603fbc86cb13d6...
    PERMIT_KEY 02954e5b20976d4a308e003917740e01e18893275a462f6799...

louis@louis-VirtualBox:~$ sawtooth identity role create transactor policy_batch
Role committed in 0.51717 sec
louis@louis-VirtualBox:~$ sawtooth identity role list
transactor: policy_batch
louis@louis-VirtualBox:~$ sawtooth block list
NUM  BLOCK_ID                                                      BATS  TXNS  SIGNER
4    d6984a442e0bab4a1803724452f3b9739fca710b2b74eb3e60f843074cac97bc5d548964c7c
0f657e6dcbf773b9ff8f21720d3beb5f37921939d30b09d3fbbda  1     1     03561c...
3    a00389e189cc5c6a39872e37c365622f9e944dbcead2af641aa7523eb8ccd776751668d5048
a64f0d8da8779c54d9ba20984d5ef5ef1e42b4f99fff227f5ab93  1     1     03374d...
2    3cbdd323d43cd8559283c0f85c77268770fdaac118660f1509d392cb930b063203e474659a1
c89fa6c68161b1160bcad8f1d3a385c7c82d81393534652f1e8dd  1     1     03a315...
1    3fd6218652d7825d68d1ea98160b7936d28fa0a7758d3d1c0eeae9f04fc950742fe1d080f7e
2be584782f079399f0df2bc1568cdd368e650fd17bebc311e15be  1     1     03561c...
0    499e7bac4815c66cfb636224d44dd487db1e6c2ec4023404331a298e2de385ac150b5102f61
1d19f1f64c603d9b3aaaa307a5366f3d9548194195f0afa480616  2     4     03374d...
louis@louis-VirtualBox:~$
```

Figure 11.11.1: Transactor key permissioning

```
louis@louis-VirtualBox:~$ sawtooth block list
NUM  BLOCK_ID                                                        BATS  TXNS  SIGNER
5    948c51040b8216102493bcbdbbf81146a01977133ae02d87a9978005b4c26d7c2799d5ded37
d99232645f6e9ca65766f1cfac0feb475bc2079084ed4c4be15c4  1     1     03374d...
4    d6984a442e0bab4a1803724452f3b9739fca710b2b74eb3e60f843074cac97bc5d548964c7c
0f657e6dcbf773b9ff8f21720d3beb5f37921939d30b09d3fbbda  1     1     03561c...
3    a00389e189cc5c6a39872e37c365622f9e944dbcead2af641aa7523eb8ccd776751668d5048
a64f0d8da8779c54d9ba20984d5ef5ef1e42b4f99fff227f5ab93  1     1     03374d...
2    3cbdd323d43cd8559283c0f85c77268770fdaac118660f1509d392cb930b063203e474659a1
c89fa6c68161b1160bcad8f1d3a385c7c82d81393534652f1e8dd  1     1     03a315...
1    3fd6218652d7825d68d1ea98160b7936d28fa0a7758d3d1c0eeae9f04fc950742fe1d080f7e
2be584782f079399f0df2bc1568cdd368e650fd17bebc311e15be  1     1     03561c...
0    499e7bac4815c66cfb636224d44dd487db1e6c2ec4023404331a298e2de385ac150b5102f61
1d19f1f64c603d9b3aaaa307a5366f3d9548194195f0afa480616  2     4     03374d...
louis@louis-VirtualBox:~$ we set -name test5 -listId 5 4 1 -listConsumption 5 7
6
Error: Error 400: Bad Request
louis@louis-VirtualBox:~$
```

Figure 11.11.2: Error when commit

11.12 Validator key permissioning

The validator key permissioning also works with a pair role-policy. However, this time, the keys needed are the validator public keys (**etc/sawtooth/keys/validator.pub**). Again the following command creates a policy that allows all the keys.

```
\$ sawtooth identity policy create policy_2 "PERMIT_KEY *"
```

The next step is to bind the network role with a given policy. The network role means that if a validator receives a peer request from a node whose validator public key is not permitted by the policy, the message is dropped and an "AuthorizationViolation" message is returned. After that, the connection is closed.

```
\$ sawtooth identity role create network policy_2
```

The figure 11.12.1 shows the creation of a new policy for the network permissioning. Then the network role is set to this policy. Note that, as for the transaction key permissioning, it is a on chain setting and so blocks are added to the blockchain when the validator key permissioning is performed.
The figure 11.12.2 shows that a new node that wants to connect to an existing network can not have access to the blockchain if it is not in the network policy. When a non authorized block tries to access the blockchain this block receives an error

503 message. One block in the blockchain receives also an "AuthorizationViolation" message to warn that an unauthorized node tried to access the blockchain.

```
louis@louis-VirtualBox:~$ sawtooth identity policy create policy_network "PERMIT
_KEY 03753418d9f033baa5196b44c9aadb993801ace4a8a24b8cf394ae5a22bebe032f" "PERMIT
_KEY 03561c08081e445fd51f7a6752ea21f0e47930c20fed81f287858730d1651c640f" "PERMIT
_KEY 03a315e96c823fbdcae252c6553475327406f4f766e7bf5d3bf064892dc36d9bcc" "PERMIT
_KEY 03374d62843cdc25649a10f01ecb4351adf212e27fc73e586e847819029c50aa2e"
Policy committed in 0.656464 sec
louis@louis-VirtualBox:~$ sawtooth identity policy list
policy_batch:
  Entries:
    PERMIT_KEY 033cc9c13e6a47d3c70465fb5145d05e91196acab72b19a21b...
    PERMIT_KEY 021f464aef19c2b076dc859f252e3790d7728ba90ae4f077ff...
    PERMIT_KEY 0224e12a5b349047253fb706ce6741b17307603fbc86cb13d6...
    PERMIT_KEY 02954e5b20976d4a308e003917740e01e18893275a462f6799...

policy_network:
  Entries:
    PERMIT_KEY 03753418d9f033baa5196b44c9aadb993801ace4a8a24b8c...
    PERMIT_KEY 03561c08081e445fd51f7a6752ea21f0e47930c20fed81f2...
    PERMIT_KEY 03a315e96c823fbdcae252c6553475327406f4f766e7bf5d...
    PERMIT_KEY 03374d62843cdc25649a10f01ecb4351adf212e27fc73e58...

louis@louis-VirtualBox:~$ sawtooth identity role create network policy_network
Role committed in 0.573093 sec
louis@louis-VirtualBox:~$ sawtooth identity role list
network: policy_network
transactor: policy_batch
```

Figure 11.12.1: Network permissioning

```
louis@louis-VirtualBox:~$ sawtooth block list
Error: http://localhost:8008: 503 Service Unavailable
louis@louis-VirtualBox:~$
```

Figure 11.12.2: Error when access to the network

```
[2021-03-10 09:41:16.767 WARNING  handlers] Received AuthorizationViolation from
 7f27482f9612c4254513d7d2a433f7eae07486cd1701adcbb5279902de5c1a6557d96116f4eb73f
ed767398f6ee3f0f07c505a5aca6605df24e2150632ad83d9
```

Figure 11.12.3: Authorization violation

11.13 Stop Sawtooth components

Use this procedure if you need to stop or reset the Sawtooth environment for any reason.

To stop the Sawtooth components, you can enter CTRL-c in all the terminal windows. A single CTRL-c does a graceful shutdown. If you prefer not to wait, you can enter multiple CTRL-c characters to force the shutdown.
To completely reset the Sawtooth environment and start over from the beginning, add these steps:

- To delete the blockchain data, remove all files from /var/lib/sawtooth.
- To delete the Sawtooth logs, remove all files from /var/log/sawtooth.
- To delete the Sawtooth keys, remove the key files /etc/sawtooth/keys/validator \.* and the keys contained in /home/yourname/.sawtooth/keys.

11.14 NAT: global explanation

This section describes the deployment of our solution if the different nodes are not on the same local network. This method uses a NAT (Network address translation) and more especially the IP masquerading. This technique is used to hide an entire IP address space, usually consisting of private IP addresses, behind a single IP address in another, usually public address space. The hidden addresses are changed into a single (public) IP address as the source address of the outgoing IP packets so they appear as originating not from the hidden host but from the routing device itself. Because of the popularity of this technique to conserve IPv4 address space, the term NAT has become virtually synonymous with IP masquerading [68]. This concept is represented in the figure 11.14.1. On this figure, the public addresses are represented with the green color and the private addresses are represented with the red color. The goal of the NAT is to reach the three local hosts through the router.

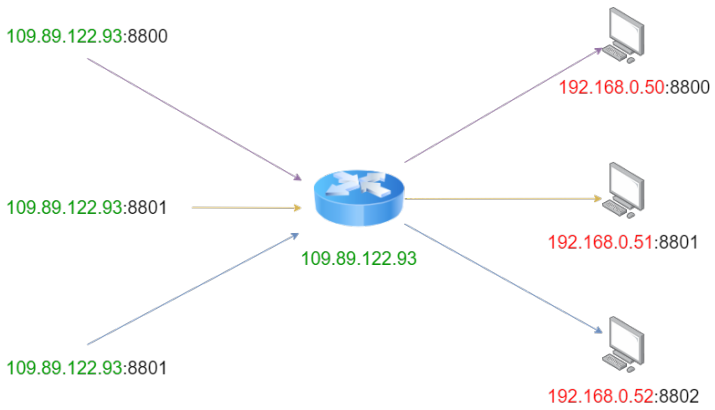


Figure 11.14.1: NAT

In this example, the public IP address of the router is 109.89.122.93. This address can be reached by anybody on internet. On the other hand, the private addresses can not directly be reached but are known by the router. In order to access a local host, the router is used as a relay. If someone (A) wants to communicate with the first machine (192.168.0.50:8800), the IP address that A uses in order to reach this first machine is $\underbrace{109.89.122.93}_{public\ IP\ of\ the\ router} : \underbrace{8800}_{port}$. Thanks to the NAT configuration, the router knows that the packets coming from the port 8800 have to be redirected to the address 192.168.0.50 on the port 8800.

11.15 NAT: deployment

This section details how to perform the NAT configuration on the router. This is explained for two internet providers : Voo and Proximus.

11.15.1 NAT for Voo

The first thing to do is to connect to the router. This can be done by connecting to http://192.168.0.1 (see fig 11.15.1). The user name is 'voo' and the password is printed under your modem.

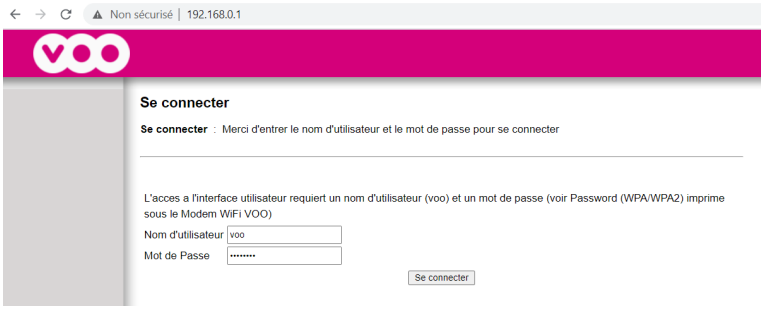


Figure 11.15.1: Home page of VOO

After going to Router -> NAT, you will arrive to a page like the figure 11.15.2.

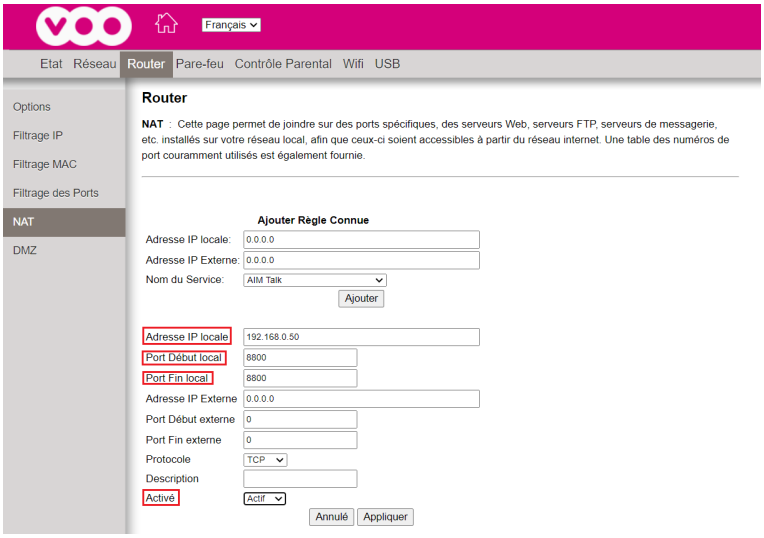


Figure 11.15.2: Configuration of the NAT

In order to configure the NAT, you only have to fill the "Adresse IP locale", "Port début local", "Port fin local" and "Activé" fields.
The "Adresse IP locale" field can be filled with the local IP address of the local machine. It can be found with the command "hostname -I" (see figure 11.15.3)

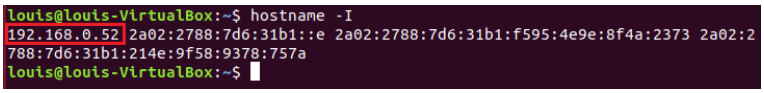


Figure 11.15.3: Local IP of a machine

The "Port début local" and "Port fin local" can be filled with the port on which you want your local machine to listen. Be careful to not use a port that is already used. At the end, do not forget to put "actif" on the "Activé" field.
The NAT configuration corresponding to the figure 11.14.1 is represented in the figure 11.15.4.

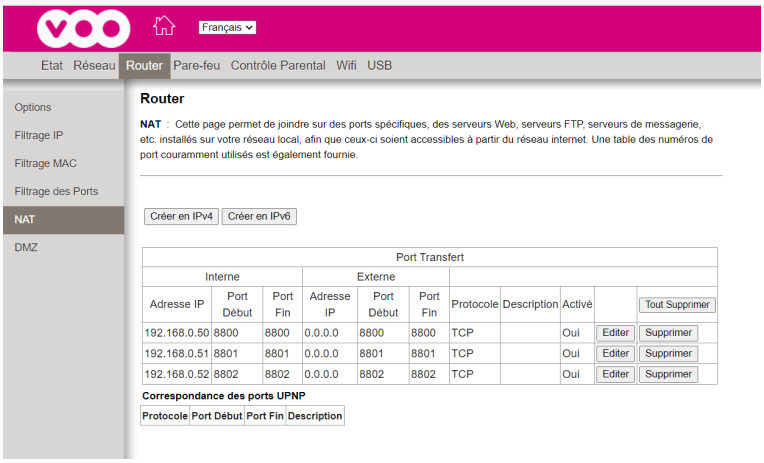


Figure 11.15.4: NAT configuration

In the frame of the deployment of our solution, we propose to use NAT in order to reach the different nodes (if they are not place on the same local network). After the deployment of the NAT, do not forget to correctly fill the peers and endpoint settings of the **validator.toml** file (see section 11.4). This time, the public address of the router should be used for those settings. The endpoint and peers parameters of the first command of the section 11.7 used to start the validator should also be adapted consequently.

11.15.2 NAT for Proximus

Access to the router configuration by connecting to http://192.168.1.1 (cf. figure 11.15.5)

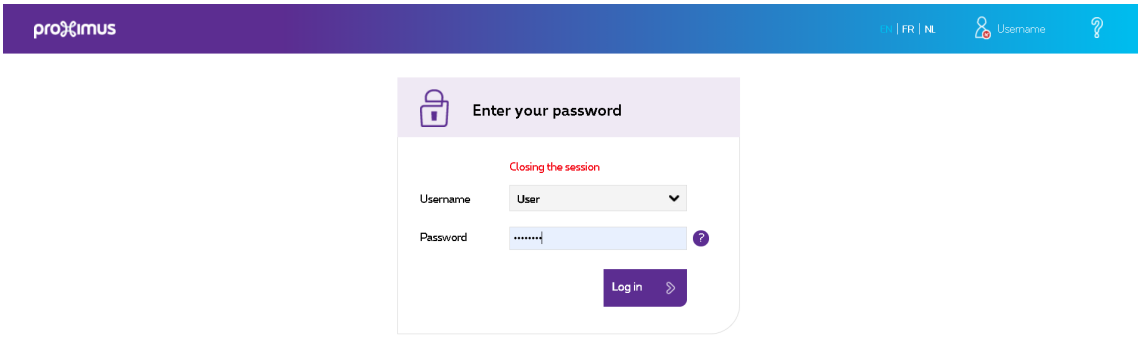


Figure 11.15.5: Home page of Proximus

Go to Access control - Port Mapping to configure the NAT. The figure 11.15.6

shows how to map a port.

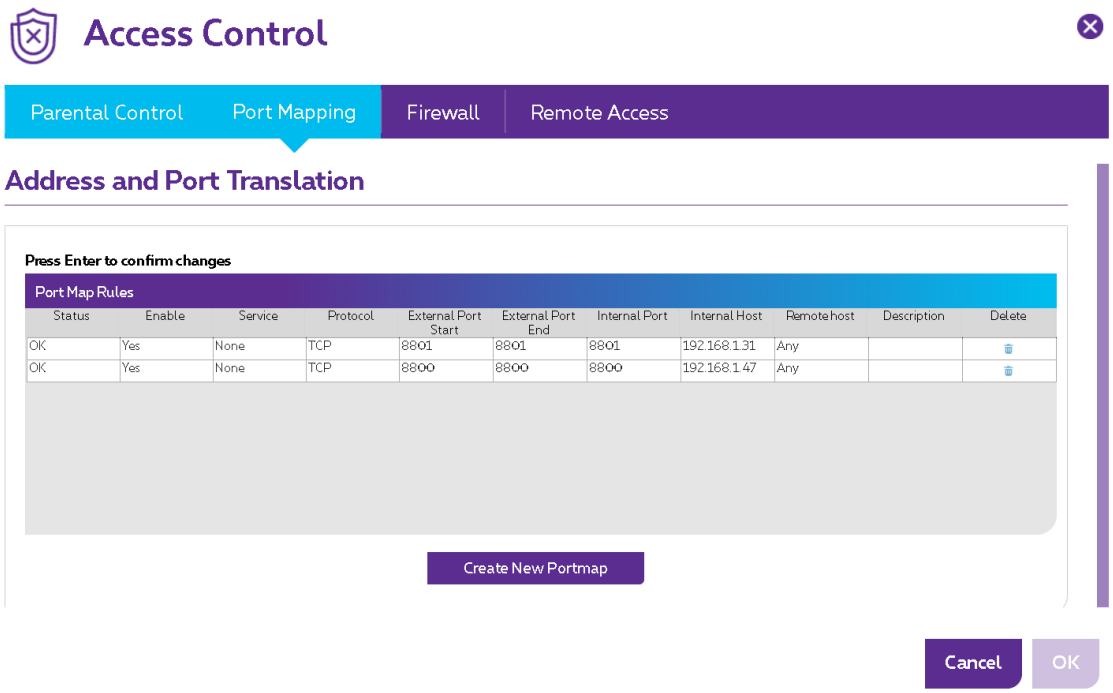


Figure 11.15.6: Port mapping

Bibliography

[1] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance and Proactive Recovery”. English. In: *ACM transactions on computer systems* 20.4 (2002), pp. 398–461. ISSN: 0734-2071.

[2] J.-J Quisquater Y. Deswarte and A. Saidane. “Remote integrity checking”. In: *Integrity and International Control in Information System VI* (2004), pp. 1–11.

[3] Ari Juels and Burton S. Kaliski Jr. “Pors: Proofs of retrievability for large files”. English. In: 2007, pp. 584–597.

[4] Satoshi Nakamoto. “Bitcoin : A peer to peer electronic cash system”. In: *https://bitcoin.org/bitcoin.pdf* (2008). Accessed: 06/04/2021.

[5] Q. Wang et al. “Enabling public verifiability and data dynamics for storage security in cloud computing”. In: *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS 2009)* (2009), pp. 355–370.

[6] Joseba Jimeno et al. “Architecture of a microgrid energy management system”. English. In: *European transactions on electrical power* 21.2 (2011), pp. 1142–1158. ISSN: 1430-144X.

[7] George Kariniotakis, Aris Dimeas, and Frank Van Overbeeke. “Pilot sites: Success stories and learnt lessons”. English. In: 2013, pp. 206–274. ISBN: 9781118720684. DOI: 10.1002/9781118720677.ch06.

[8] M. Mihaylov et al. “NRGcoin: Virtual currency for trading of renewable energy in smart grids”. In: *11th International Conference on the European Energy Market (EEM14)*. 2014, pp. 1–6. DOI: 10.1109/EEM.2014.6861213.

[9] Jin Li et al. “OPoR: Enabling proof of retrievability in cloud computing with resource-constrained devices”. English. In: *IEEE transactions on cloud computing* 3.2 (2015), pp. 195–205. ISSN: 2168-7161.

[10] H. Lin S.Shen and W. Tzeng. “An effective integrity check scheme for secure erasure code based storage systems”. In: *IEEE transactions on reliability* 64.3 (2015), pp. 840–851.

[11] Melanie Swan. *Blockchain: blueprint for a new economy*. English. Sebastopol: O'Reilly, 2015. ISBN: 1491920491.

[12] B. P. Koirala et al. "Local Alternative for Energy Supply: Performance Assessment of Integrated Community Energy Systems". English. In: *Energies (Basel)* 9.12 (2016), p. 981. ISSN: 1996-1073.

[13] Andrija Goranovic et al. "Blockchain applications in microgrids: An overview of current projects and concepts". English. In: vol. 2017-. 2017, pp. 6153–6158. ISBN: 9781538611272;1538611279;

[14] X. Liang et al. "ProvChain: A Blockchain-Based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2017, pp. 468–477. DOI: 10.1109/CCGRID.2017.8.

[15] B. Liu et al. "Blockchain Based Data Integrity Service Framework for IoT Data". In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 468–475. DOI: 10.1109/ICWS.2017.54.

[16] Esther Mengelkamp et al. "Designing microgrid energy markets: A case study: The Brooklyn Microgrid". English. In: *Applied energy* 210 (2018), pp. 870–880. ISSN: 0306-2619.

[17] Ivica Nikolic et al. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale". English. In: (2018).

[18] Jiani Wu and Nguyen K. Tran. "Application of blockchain technology in sustainable energy systems: An overview". English. In: *Sustainability (Basel, Switzerland)* 10.9 (2018), p. 3067. ISSN: 2071-1050.

[19] Jiani Wu and Nguyen Khoi Tran. "Application of Blockchain Technology in Sustainable Energy Systems: An Overview". In: *Sustainability* 10.9 (2018). ISSN: 2071-1050. DOI: 10.3390/su10093067. URL: https://www.mdpi.com/2071-1050/10/9/3067.

[20] D. Yue et al. "Blockchain Based Data Integrity Verification in P2P Cloud Storage". In: *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 2018, pp. 561–568. DOI: 10.1109/PADSW.2018.8644863.

[21] Merlinda Andoni et al. "Blockchain technology in the energy sector: A systematic review of challenges and opportunities". English. In: *Renewable and sustainable energy reviews* 100 (2019), pp. 143–174. ISSN: 1364-0321.

[22] Michèle Finck. *Blockchain and the general data protection regulation: Can distributed ledgers be squared with European data protection law?* English. Luxembourg: Publications Office, 2019. ISBN: 9789284650446;9284650445;

[23] Renaud H. *Fonctionnement d'un smart contract sur Ethereum.* BitConseil. 2019. URL: https://bitconseil.fr/smart-contract-ethereum/. (accessed: 08/02/2021).

[24] Rui Hu et al. "A survey on data provenance in IoT". English. In: *World wide web (Bussum)* 23.2 (2019), pp. 1441–1463. ISSN: 1386-145X.

[25] E. A. Kanimozhi, M. Suguna, and S. Mercy Shalini. "Immediate Detection of Data Corruption by Integrating Blockchain in Cloud Computing". In: *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN).* 2019, pp. 1–4. DOI: 10.1109/ViTECoN.2019.8899394.

[26] Swagatika Sahoo et al. "A Hierarchical and Abstraction-Based Blockchain Model". In: *Applied Sciences* 9.11 (2019). ISSN: 2076-3417. DOI: 10.3390/app9112343. URL: https://www.mdpi.com/2076-3417/9/11/2343.

[27] Swagatika Sahoo et al. "A hierarchical and abstraction-based blockchain model". English. In: *Applied sciences* 9.11 (2019), p. 2343. ISSN: 2076-3417.

[28] K. Sharma and D. Jain. "Consensus Algorithms in Blockchain Technology: A Survey". In: *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT).* 2019, pp. 1–7. DOI: 10.1109/ICCCNT45670.2019.8944509.

[29] Hai Wang et al. "An overview of blockchain security analysis". English. In: vol. 970. 2019, pp. 55–72.

[30] Stephan Cejka et al. "A blockchain-based privacy-friendly renewable energy community". English. In: 2020, pp. 95–103. ISBN: 9897584188;9789897584183;

[31] Huawei Mei and Gen Li. "Energy efficiency of microgrid and the application of blockchain in microgrid". English. In: *IOP conference series. Earth and environmental science* 558.5 (2020), p. 52046. ISSN: 1755-1307;1755-1315;

[32] Yustus Oktian, Sanggon Lee, and Hoon Lee. "Hierarchical Multi-Blockchain Architecture for Scalable Internet of Things Environment". In: *Electronics* 9 (June 2020), p. 1050. DOI: 10.3390/electronics9061050.

[33] Shermin Voshmgir. *What is a blockchain ?* Blockchainhub Berlin. 2020. URL: https://blockchainhub.net/blockchain-intro/%5C#:%5C~:text=A%5C%20Blockchain%5C%20protocol%5C%20operates%5C%20on,a%5C%20middleman%5C%20though%5C%20machine%5C%20consensus.%5C&text=This%5C%20ledger%5C%20runs%5C%20on%5C%20a,(P2P)%5C%20network%5C%20of%5C%20computers!. (accessed: 29/03/2021).

[34] Qiang Wang and Min Su. “Integrating blockchain technology into the energy sector —from theory of blockchain to research and application of energy blockchain”. English. In: *Computer science review* 37 (2020), p. 100275. ISSN: 1574-0137.

[35] Zhuoli Zhao et al. “Energy Transaction for Multi-Microgrids and Internal Microgrid Based on Blockchain”. English. In: *IEEE access* 8 (2020), pp. 144362–144372. ISSN: 2169-3536.

[36] Binance academy. *Tendermint Explained*. Academy. URL: https://academy.binance.com/en/articles/tendermint-explained. (accessed: 07/02/2021).

[37] Dan Anderson. *Hyperledger Sawtooth Blockchain Performance Metrics with Grafana*. Intel Corporation. URL: https://www.hyperledger.org/blog/2019/01/25/hyperledger-sawtooth-blockchain-performance-metrics-with-grafana. (accessed: 12/05/2021).

[38] *Blockchain Architecture*. Cosmos. URL: https://docs.cosmos.network/master/intro/sdk-app-architecture.html. (accessed: 07/02/2021).

[39] *Blockchain privees, publique et a consortium - Quelles sont les differences ?* Binance academy. URL: https://academy.binance.com/fr/articles/private-public-and-consortium-blockchains-whats-the-difference%5C#consortium-blockchains.

[40] *Blockchain protocols - Corda*. Chainstack. URL: https://chainstack.com/protocols/corda/. (accessed: 07/02/2021).

[41] *Blockchain protocols - Fabric*. Chainstack. URL: https://chainstack.com/protocols/fabric/. (accessed: 07/02/2021).

[42] *Blockchain protocols - multichain*. Chainstack. URL: https://chainstack.com/protocols/multichain/. (accessed: 07/02/2021).

[43] *Blockchain protocols - quorum*. Chainstack. URL: https://chainstack.com/protocols/quorum/. (accessed: 07/02/2021).

[44] Floriane Bobée. *Proof of stake : Définition et explication*. bitconseil. URL: https://bitconseil.fr/proof-of-stake-definition-explication/. (accessed: 09/02/2021).

[45] Software brussels. *WeSmart*. URL: https://software.brussels/listing/wesmart/. (accessed: 20/03/2021).

[46] Cristina Criddle. *Bitcoin consumes 'more electricity than Argentina'*. BBC. URL: https://www.bbc.com/news/technology-56012952. (accessed: 07/04/2021).

[47] *Cryptography Digital signatures*. Tutorials Point. URL: https://www.tutorialspoint.com/cryptography/cryptography%5C_digital%5C_signatures.htm. (accessed: 08/02/2021).

[48] Brian Curran. *What is Proof of Elapsed Time Consensus? (PoET) Complete Beginner's Guide*. Blockonomi. URL: https://blockonomi.com/proof-of-elapsed-time-consensus/. (accessed: 09/04/2021).

[49] Bruno D. *Brooklyn's latest craze: making your own electric grid*. politico. URL: https://www.politico.com/magazine/story/2017/06/15/how-a-street-in-brooklyn-is-changing-the-energy-grid-215268/. (accessed: 06/04/2021).

[50] *Decentralized Vs. Centralized: A Detailed Comparison*. 101 Blockchains. URL: https://101blockchains.com/decentralized-vs-centralized/. (accessed: 05/05/2021).

[51] *DPoS*. BitcoinWiki. URL: https://en.bitcoinwiki.org/wiki/DPoS. (accessed: 07/04/2021).

[52] *Error correction code*. Wikipedia. URL: https://en.wikipedia.org/wiki/Error%5C_correction%5C_code. (accessed: 13/04/2021).

[53] Greeneum. *GREENEUM Global Energy Networks Whitepaper*. URL: https://static1.squarespace.com/static/5a3de333c027d8dd95183ca7/t/5a750b678165f51a34ddeb6e/1517620075918/Greeneum+Whitepaper+v2.pdf. (accessed: 13/05/2021).

[54] 2020 Hasib Anwaron July 20. *Hyperledger Sawtooth Vs Fabric : How are they different ?* 101blockchains. URL: https://101blockchains.com/hyperledger-sawtooth-vs-fabric/. (accessed: 07/02/2021).

[55] Parikshit Hooda. *practical Byzantine Fault Tolerance(pBFT)*. Geeksforgeeks. URL: https://www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/. (accessed: 27/04/2021).

[56] Parikshit Hooda. *Sybil Attack*. Geeksforgeeks. URL: https://www.geeksforgeeks.org/sybil-attack/%5C#:%5C~:text=Sybil%5C%20Attack%5C%20is%5C%20a%5C%20type,authority%5C%2Fpower%5C%20in%5C%20reputation%5C%20systems.. (accessed: 10/04/2021).

[57] *How BondEvalue launched the world's first fractional bond exchange with Hyperledger Sawtooth*. Hyperledger. URL: https://www.hyperledger.org/learn/publications/bondevalue-case-study. (accessed: 20/05/2021).

[58] *How ScanTrust Brought Transparency to the Supply Chain with Hyperledger Sawtooth*. Hyperledger. URL: https://www.hyperledger.org/learn/publications/scantrust-case-study. (accessed: 20/05/2021).

[59] *Hyperledger Sawtooth Blockchain Security (Part Two)*. Hyperledger. URL: https://www.hyperledger.org/blog/2018/11/23/hyperledger-sawtooth-blockchain-security-part-two. (accessed: 15/04/2021).

[60] *Introduction*. Sawtooth. URL: https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html. (accessed: 10/02/2021).

[61] Gwyneth Iredale. *Public Vs Private Blockchain: How Do They Differ?* https://101blockchains.com/public-vs-private-blockchain/. Accessed: 10/04/2021.

[62] André Joffre. *Laeken (Bruxelles) : partager son énergie solaire, le nouveau projet signé Greenbizz*. Youtube. URL: https://www.youtube.com/watch?v=cL0mDKjjBuI. (accessed: 20/03/2021).

[63] *Jouliette at De Ceuvel*. URL: https://www.jouliette.net/. (accessed: 13/05/2021).

[64] Julian Martinez. *Understanding Proof of Stake: The Nothing at Stake Theory*. medium. URL: https://medium.com/coinmonks/understanding-proof-of-stake-the-nothing-at-stake-theory-1f0d71bc027. (accessed: 07/04/2021).

[65] Demiro Massessi. *Public Vs Private Blockchain In A Nutshell*. Medium. URL: https://medium.com/coinmonks/public-vs-private-blockchain-in-a-nutshell-c9fe284fa39f. (accessed: 10/04/2021).

[66] Zdenek Pekarek Max N. Luke Stephen J. Lee and Anna Dimitrova. *Blockchain in Electricity: a Critical Review of Progress to Date*. Nera economic consulting. URL: https://cdn.eurelectric.org/media/3115/paper1_blockchain_eurelectric-h-BA73FBD9.pdf. (accessed: 13/05/2021).

[67] Morisander. *The biggest smart contract hacks in history or how to endanger up to US $2.2 billion*. Medium. URL: https://medium.com/solidified/the-biggest-smart-contract-hacks-in-history-or-how-to-endanger-up-to-us-2-2-billion-d5a72961d15d. (accessed: 06/05/2021).

[68] *Network address translation*. Widipedia. URL: https://en.wikipedia.org/wiki/Network%5C_address%5C_translation. (accessed: 10/03/2021).

[69] David Etheridge Norbert Schwieters Jeroen van Hoof and Axel von Perfall. *Blockchain – an opportunity for energy producers and consumers?* pwc. URL: https://www.pwc.com/gx/en/industries/assets/pwc-blockchain-opportunity-for-energy-producers-and-consumers.pdf. (accessed: 13/05/2021).

[70] Team InnerQuest Online and InnerQuest Online. *How Does a Blockchain Prevent Double-Spending of Bitcoins?* medium. URL: https://medium.com/innerquest-online/how-does-a-blockchain-prevent-double-spending-of-bitcoins-fa0ecf9849f7%5C#:%5C~:text=In%5C%20summary%5C%2C%5C%20the%5C%20blockchain%5C%20prevents,and%5C%20impossible%5C%20to%5C%20tamper%5C%20with.. (accessed: 09/02/2021).

[71] *PBFT Architecture*. Hyperledger Sawtooth. URL: https://sawtooth.hyperledger.org/docs/pbft/releases/latest/architecture.html. (accessed: 10/03/2021).

[72] ProgrammerSought. *Understanding distributed consistency: Byzantine fault tolerance and PBFT*. programmersought. URL: https://www.programmersought.com/article/40734411868/%5C#PBFT%5C_%5C_47. (accessed: 27/04/2021).

[73] *Proof of authority*. Wikipedia. URL: https://en.wikipedia.org/wiki/Proof%5C_of%5C_authority. (accessed: 08/04/2021).

[74] *Proof of Authority: consensus model with Identity at Stake*. Medium. URL: https://medium.com/poa-network/proof-of-authority-consensus-model-with-identity-at-stake-d5bd15463256. (accessed: 08/04/2021).

[75] *Radix tree*. Wikipedia. URL: https://en.wikipedia.org/wiki/Radix%5C_tree. (accessed: 05/03/2021).

[76] Pete Rizzo. *Tendermint Exploring Possible Public Blockchain Launch*. coindesk. URL: https://www.coindesk.com/tendermint-public-blockchain. (accessed: 07/02/2021).

[77] Peter Van Roy. *Node Behavior (failures)*. Slide from LSINF2345 - Languages and Algorithms for Distributed Applications, 2020.

[78] *Sawtooth documentation*. Sawtooth. URL: https://sawtooth.hyperledger.org/docs/core/releases/latest/app%5C_developers%5C_guide.html. (accessed: 07/02/2021).

[79] *Sawtotooth PBFT*. Hyperldeger. URL: https://sawtooth.hyperledger.org/docs/pbft/releases/latest/index.html. (accessed: 11/02/2021).

[80] Logan Seeley. *Introduction to Sawtooth PBFT*. Hyperldger. URL: https://www.hyperledger.org/blog/2019/02/13/introduction-to-sawtooth-pbft. (accessed: 11/02/2021).

[81] Toshendra Kumar Sharma. *Public VS. Private Blockchain : A Comprehensive Comparison*. blockchain concil. URL: https://www.blockchain-council.org/blockchain/public-vs-private-blockchain-a-comprehensive-comparison/. (accessed: 10/04/2021).

[82] Meng Shen, Liehuang Zhu, and Ke Xu. *Blockchain: Empowering Secure Data Sharing*. English. Singapore: Springer Singapore. ISBN: 9789811559389;9811559384;

[83] Todd Sobrado. *What are the six layers of Blockchain technology?* URL: https://toddsobrado.com/what-are-the-six-layers-of-blockchain-technology/. (accessed: 23/05/2021).

[84] Lucian Toma Steffen Bretzke Miguel Ponce de Leon. *RESERVE*. europa.eu. URL: https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5b5752d66%5C&appId=PPGMS. (accessed: 04/04/2021).

[85] Nick Szabo. *Smart Contract*. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html. (accessed: 08/02/2021).

[86] *Tendermint*. Tendermint. URL: https://docs.tendermint.com/master/. (accessed: 07/02/2021).

[87] *Trie*. Wikipedia. URL: https://en.wikipedia.org/wiki/Trie. (accessed: 05/03/2021).

[88] Chjango Unchained. *Tendermint Explained — Bringing BFT-based PoS to the Public Blockchain Domain*. Cosmos. URL: https://blog.cosmos.network/tendermint-explained-bringing-bft-based-pos-to-the-public-blockchain-domain-f22e274a0fdb. (accessed: 07/02/2021).

[89] *Using Ubuntu for a Sawtooth Test Network*. Hyperledger. URL: https://sawtooth.hyperledger.org/docs/core/releases/latest/app%5C_developers%5C_guide/ubuntu%5C_test%5C_network.html. (accessed: 09/03/2021).

[90] Victor. *What is Practical Byzantine Fault Tolerance (PBFT)?* Crushcrypto. URL: https://crushcrypto.com/what-is-practical-byzantine-fault-tolerance/. (accessed: 09/04/2021).

[91] *Vous êtes-vous déjà demandé comment les Bitcoins (et autres crypto-monnaies) marchaient en réalité ?* 3Blue1Brown. URL: https://www.youtube.com/watch?v=bBC-nXj3Ng4. (accessed: 09/02/2021).

[92] *WeSmart et son projet de communauté « Greenbizz.energy » font partie des lauréats BeCircular !* Greenbizz. URL: https://www.greenbizz.brussels/fr/wesmart-et-son-projet-de-communaute-greenbizz-energy-font-partie-des-laureats-becircular/. (accessed: 20/03/2021).

[93] *What is a hash function? Definition, usage, and examples*. Digital guide IONOS. URL: https://www.ionos.com/digitalguide/server/security/hash-function/. (accessed: 08/02/2021).

[94] *What is Public-key Cryptography?* GlobalSign. URL: https://www.globalsign.com/en/ssl-information-center/what-is-public-key-cryptography. (accessed: 08/02/2021).

[95] *When Hyperledger Sawtooth Met Kubernetes - Simplifying Enterprise Blockchain Adoption*. Hyperledger. URL: https://www.hyperledger.org/learn/publications/kubernetes-case-study. (accessed: 20/05/2021).

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**