Politecnico di Torino
UCL - Centre for Blockchain Technologies

Blockchain in the Bitcoin Era

Gaetano Mondelli

Submitted in part fulfilment of the requirements for the degree of
Laurea Magistrale in Ingegneria Informatica (Computer Engineering)
of the Politecnico di Torino, London (Supervisors: P.Tasca - E. Piccolo) August 2017

‘***To Evariste Galois***’

Abstract

The spread of DLT, i.e. distributed ledger technologies, and their applications besides finance can be an opportunity for solving problems we were not able to do before. While giving a broad range of benefits, the fast-paced environment of distributed ledger technologies lacks on scalablity, extensibility, interoperability and the ability to upgrade without side effects. Upgrade a blockchain means to create a fork and since user's migration cannot be forced this situation is difficult to handle. All systems requires update to meet the technology requirements and the trends' needs. Sometimes upgrade are mandatory because of security flaws . This incapability to adapt himself makes hard the adoption of blockchain by companies and institutions despite the huge benefit they could gain. Many proposals try to address some of the issue of this technology. However it is difficult to foresee the needs and future security issues therefore it is hard to choose the proper DLT. The vast majority of existing technologies that are trying to solve scalability and extensibility issues defines a new standard to allow add new blocks and extend their functionality.

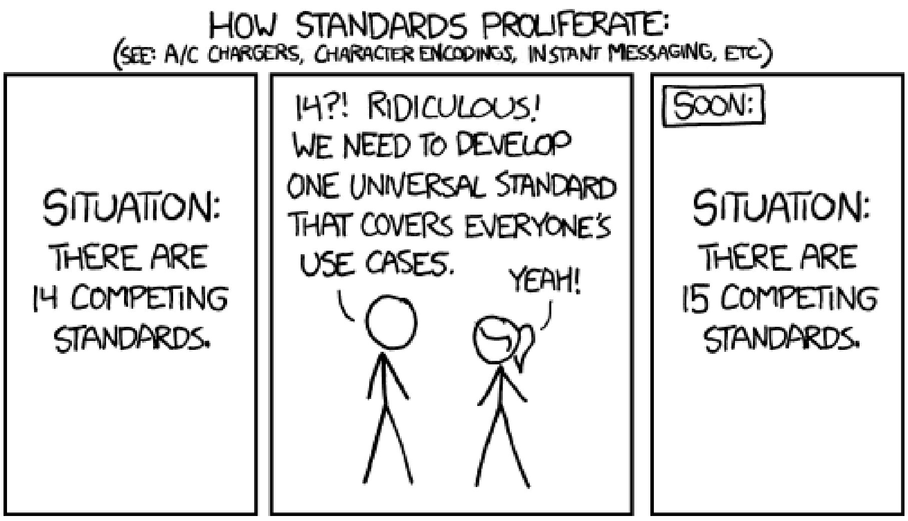

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Figure 1: xkcd comic 927 on Standards. We are facing the same situation for allow cross blockchains communications

However, these standards are strictly related to the functionality and the type of scale we require in this moment and in the long run they will not be able to satisfy technology and functional needs. This paper will introduce the basic concepts of blockchains and cryptographic data structures like Hash pointers and Merkle trees. We will start the definition of the hash

pointers to the properties guaranteed by them. We will define how to build a cryptocurrency using these tools. After having seen the anatomy of a blockchain, we will analyse their different functionalities. We will try to abstract them to understand what are the key aspects to promote the blockchain technology to the next level and make it a safe choice designing a system. I have been always interested in distributed technology and blockchain. The collaboration with the UCL Centyre for Blockchain Technology gave me the opportunity to share knowledge and insights on the future of cryptocurrencies and distributed ledger.

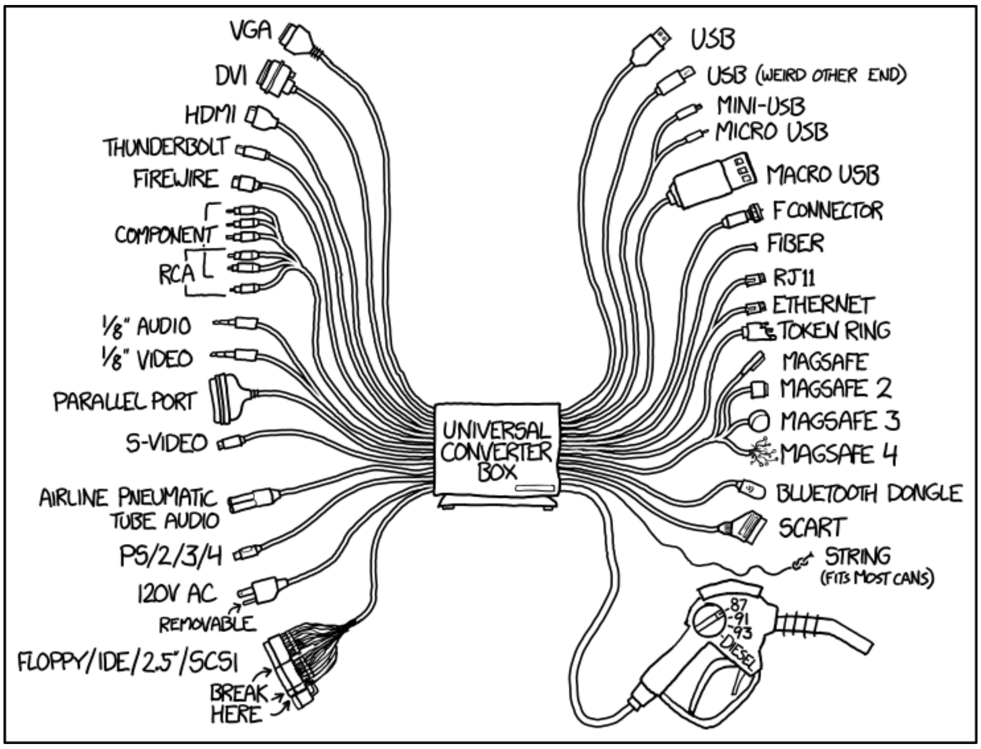


Figure 2: xkcd-1406, existing solutions create specific solutions to problems and then try to adapt them to others introducing complexity.

Contents

List of Tables

List of Figures

Chapter 1

Cryptography tools and properties

'-It's a poor sort of memory that only works backwards- the Queen remarked.'

Lewis Carroll, *Through the Looking-Glass, and What Alice Found There.*

In this chapter we will summaries the concepts and tools necessary to build a distributed ledger technology. We will use an incremental approach starting from the cryptography algorithms and their property we will build a simple cryptocurrencies. We will analyze what property are guaranteed, what are the benefit and what are drawbacks. We will address all the issues and we will describe how modern technology solves these problems. The following approach to explain the blockchain technology is the same proposed by the book Bitcoin and Cryptocurrency Technologies [AN15]. This is a good way to familiarise with the technology and we will use the same terminology in this paper.

EXPLAIN TO ME HOW BITCOIN WORKS.
AND DON'T GIVE ME THE CONDESCENDING SIMPLE VERSION FOR MANAGERS. I WANT A FULL TECHNICAL EXPLANATION.
EARLY CIVILIZATIONS HAD NO CONCEPT OF ZERO.
GO ON.
Derivative work. Original by Scott Adams. www.dilbert.com
Original © 2003 United Feature Syndicate, Inc.

Figure 1.1: Dilbert comics on how difficult can be explain the blockchain technology

1.1 Secure hash functions

Hash Functions are widespread in Computer Science. They are mathematical functions that offer the possibility to transform an object, typically a string, in an object of a smaller dimension, usually an integer or a string. In the most common scenario, they are used to build a sort of bi-unique relation between an object and a key that represents it. Since they try to map a larger set into a smaller one, all hash functions can sufferer for collisions, i.e. two different objects have the same transformations.

$$Hash(a) = Hash(b) \qquad a \neq b$$

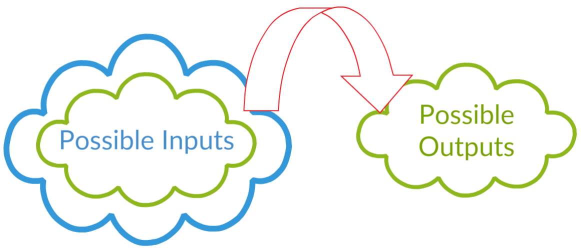


Figure 1.2: Hash input set is bigger than the output one.

Hash functions must also produce a fixed-size output, (e.g. 256 bits in the Bitcoin context). They are supposed to be efficiently solvable, therefore given a string; a hash function must be able to calculate the result in a reasonable length of time. Regarding complexity, the algorithms that implement the functions are linear to the data dimension. A cryptography hash function is a hash function that satisfies three more rules

- Collision Free property
- Hiding property "one-way"
- Puzzle-Friendly property

The collision free property means it is not possible to find two different elements that have the same hash. It is not to say that in these functions, collisions do not exist. As shown before, collisions exist, what the collision free property guarantees, is that no one can find collisions.

If someone can find two elements with the same hash, the function is considered insecure. An example of this happened in February 2017 when Google published two items that give the same output as result of the SHA-1 function [MSM17]. As long as the collision free property is valid, if two objects have the same hash, it is safe to assume that those are the same object. The hiding property is related to the possibility of finding what was the input of a hash. Imagine we split a coin and we want to communicate the resulting event by sending its hash. Since there are only two possibilities: "*head*" or "*tail*", it is easy to understand what happened by calculating the hash of each possible input or at least in a typical scenario, by calculating the inputs that are most likely to happen. What this intends to achieve is that given a $H(x)$ it is not feasible to find x To achieve this, it is necessary to consider a distribution with a high min-entropy so that no value has a bigger probability to be chosen. Given e from a high min-entropy distribution and concatenating it with x then it is unfeasible to find what x was, starting from $H(e|x)$. The hiding property allows what is called the commitment (i.e. seal a value in an envelope and later open it revealing the value). Therefore, given $H(e|message)$ it is not possible to find $message2$ that concatenate with e give the same hash.

$$NotFeasible : Hash(e|message) = Hash(e|message2) \qquad message \neq message2$$

The third property guarantee that given the e from a high min-entropy given $H(), e, y$ it is not feasible to find x such that $H(e|x) = y$. This property is used to build mathematical puzzle where there are no shortcuts rather than explore all the space of the solutions. Given a puzzle-id id, from a high min-entropy and a target y, the puzzle is to find the x such that: $H(id|e) = y$. Puzzle-friendly property means that there are no strategies better than random search in the space solutions.

1.1.1 SHA-256

There exists a lot of hash functions. The one the Bitcoin uses is SHA 256. It takes the message, and it breaks it up into 512-bit size blocks. Since the message is not necessarily a 512-bit-multiple in size, it adds some padding at the end. The padding consists of one bit followed by some zeros and a 64-bit length at the end that describes the length of the message. Then you use a compression function that takes as inputs the block message of 512 bit and a

256-bit array. For the first block, it uses a particular vector called Initial Vector (IV), and the compression function takes the 768 bit (256 bit + 512 bit) and returns a 256-bit array. This 256 array is used as input for the same compression function with the next message block till all the blocks are processed, the last 256-bit array is the hash of the message with respect to a given IV. Please consider that a function made by sub-functions that have the collision-free property, the hiding property and the puzzle friendly property, still has these three properties.

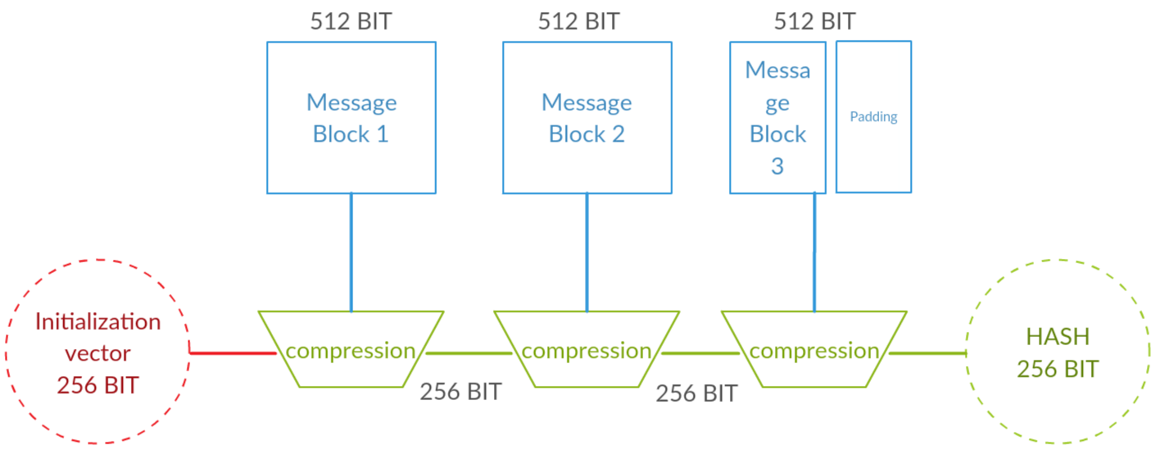


Figure 1.3: SHA 256 - Secure Hash Algorithm

1.2 Asymmetric Cryptography and signatures

Asymmetric Cryptography is required to achieve digital signatures. The Digital signature has the same role as hand-write signatures. Signatures should work in a way such that no one can replicate them and there is a way to verify that a signature is authentic. We want to satisfy the following properties with digital signatures:

- Only one entity can make a valid signature related to a particular identity
- Anyone can verify that the signature is valid
- A name is tied to a particular document
- Who signs a document cannot deny it (Non-repudiation)

Asymmetric cryptography allows reaching the three previous requirements generating a couple of tied keys. Imagine being able to forge two keys a, b that are complementary and can be used one to encrypt data generating a message that only the other one can decrypt. We keep on of the key secret and we make public the other one. If we can prove that the public one is tied with our identity, we can sign messages with the secret one making able other people check what we have signed using the public one. When you generate these keys, it is critical that the generator has been randomised, because you want to use different keys for the different identity. Not being sufficiently random can compromise the security properties. The first property is known as unforgeability, it means that no one can forge a signature on some message if he do not have the secret key. It is formulated in terms of a challenge between the key owner and a malicious adversary 1.4. The adversary has access only to a public key and has the ability to ask the secret owner to sign messages on his choice aiming to forge a signature for a document without having asked it to the private key owner. In an everyday context, a malicious adversary can collect several messages signed by the chosen victim. Hence, in this example we allow the adversary to get a signature on all the messages he wants. The fact that the adversary cannot forge signatures on new messages explain what the unforgeability property means. There are several signature schemes and among them, ECDSA (Elliptic Curve Digital Signature Algorithm) is the U.S.A. government standard, and it used by Bitcoin.

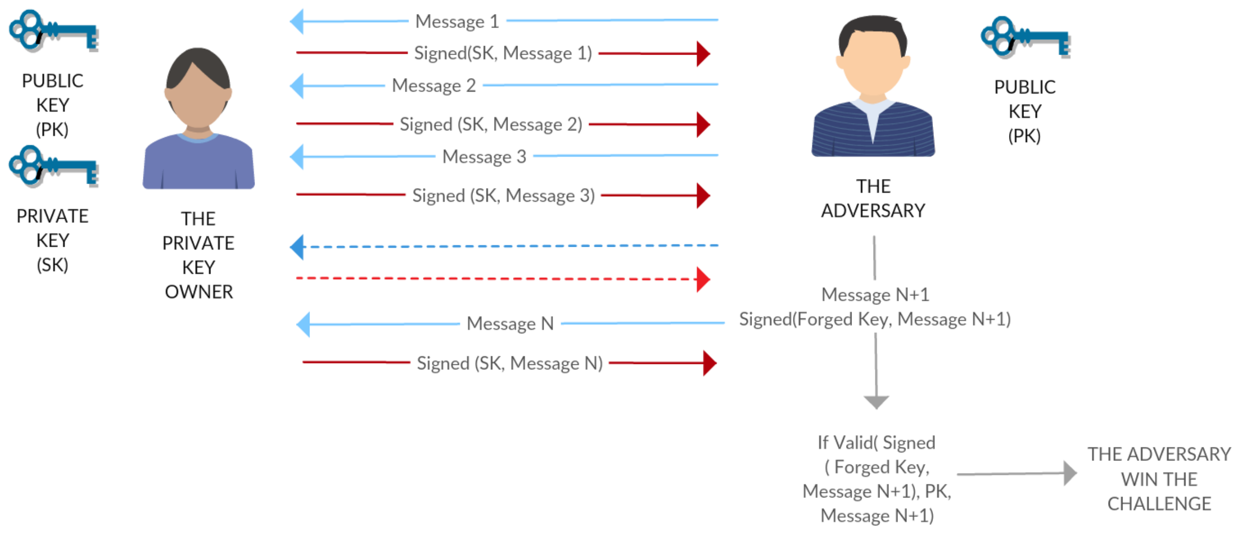


Figure 1.4: Unforgeablility Challenge.

1.3 Criptographic data strucuture

A pointer is a data structure that stores the location where an information is physically stored and allows you to retrieve a particular data. Hash pointers, also store a cryptographic hash of the stored information to let us verify that the information has not changed (integrity). It is a pointer that gives you the ability to locate its position and also allows you to check the data. Please consider that the term pointer is not something referred to a low-level data type like it would be in low level languages like C. The pointer is an information that uniquely identifies data. It acts as a key and when it is used in a dictionary it gives you back the requested information, and you are sure that no one has changed that information because calculating the hash of the retrieved information you can verify that its hash matches with the pointer. Hash pointers can be used to build all kinds of data structures. The hash pointer calculation works backwards, so the last data contains integrity information of its data and of the preceding ones. For this reason we must avoid cycles because there would be not a end we can start to compute back from. Therefore all sorts of DAGs (Directed Acyclic Graph) are allowed. In the most common scenario, a linked list that uses hash back pointers rather than regular back pointers is a blockchain.

1.3.1 Blockchain

A blockchain, is a series of blocks. Each block contains data and a pointer to the previous block. In this case, the pointer is also the hash of the previous block. If someone tries to tamper data within a block, the hash of the next block will become invalid because of the collision free property. If the adversary can change the hash pointer of the next block in a way that it match with the hash of the tampered block, then the block with the tampered hash pointer has a hash which is different from the pointer of its next block. This tampering operation will involve blocks until the adversary reaches the head that cannot be tampered because it is the only things people store and it is the way to have access to the data. The first block is the only one with no reference to a previous block, and it is called genesis block. Therefore if someone wants to prove that a data within a block belongs to a blockchain he should give the block and then check if its hash matches with the hash pointer in the next block until you reach the head. This means that the membership of a block involves in the worst case scenario to show

N blocks (N number of blocks into the blockchain).

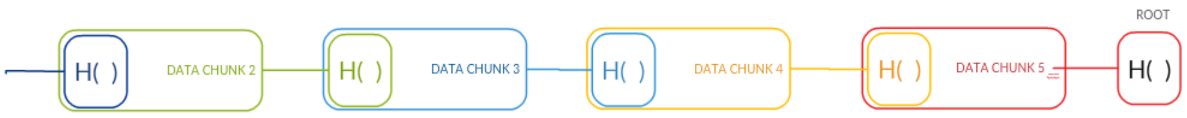


Figure 1.5: Blockchain data structure

1.3.2 Merkle Tree

A linked list is not the only interesting data structure that can be made with a hash pointer. A binary tree with hash pointers is what is commonly called Merkle tree after Ralph Merkle introduced it for the first time. Building a Merkle Tree requires to split the message up into blocks. For each consecutive pairs of blocks, the correspondent data structure has two hash pointers, on to each of these message blocks. For each pair of these data structures, consists of the same data structure with two pointers that points on this pair. These data structures are built until there are only two children to point to, and this is what is called the root. As for the blockchain, this structure protects data from tampering. If an adversary wants to tamper data, he should change the hash pointers that points on tampered data in the upper levels of the tree. Changing the hash pointers at lower levels causes the upper levels of these hash pointers to have invalid hash pointers that have to be changed as well. The adversary should change the hash pointers until he reaches the root that is what all the people need to store and therefore it is not possible to change without making all the counterparts aware about the tamper.

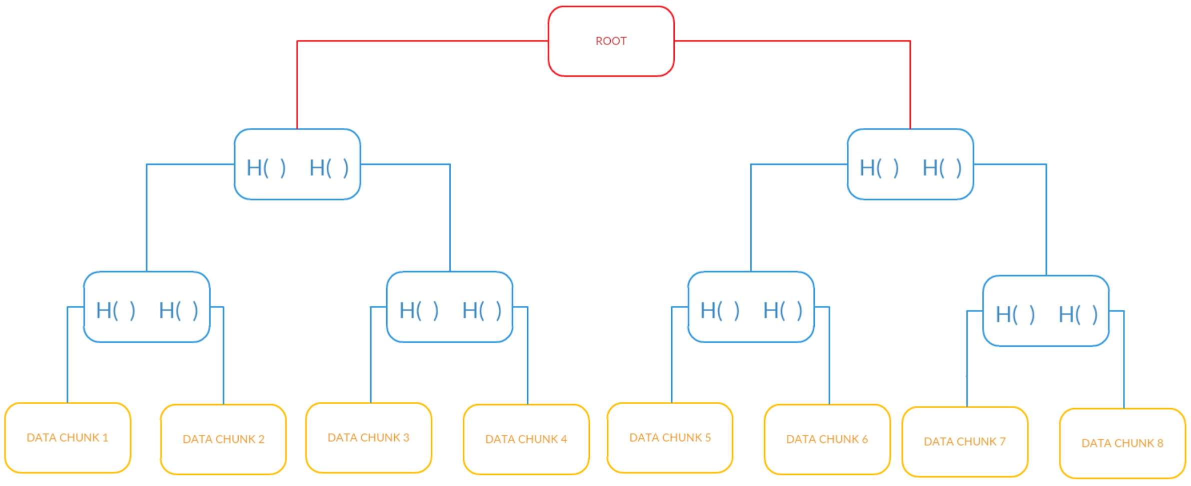


Figure 1.6: A Merkle tree structure.

Unlike the blockchain, if you want to show that a block is a member of the Merkle Tree, you need to show only the blocks in the path from the root to the blocks (the ones coloured in the representation 1.7). Who wants to verify the membership need only to check the hash and at the end compare its stored root with the calculated one from the showed blocks. In this way, we can prove membership in a smaller amount of time showing only $\log N$ blocks. There is a slightly different implementation of the Merkle tree that is called Sorted Merkle Tree. Blocks and data within the blocks are sorted in this case, to prove whether or not that an element is member or not of the tree you only need a logarithm algorithm complexity.

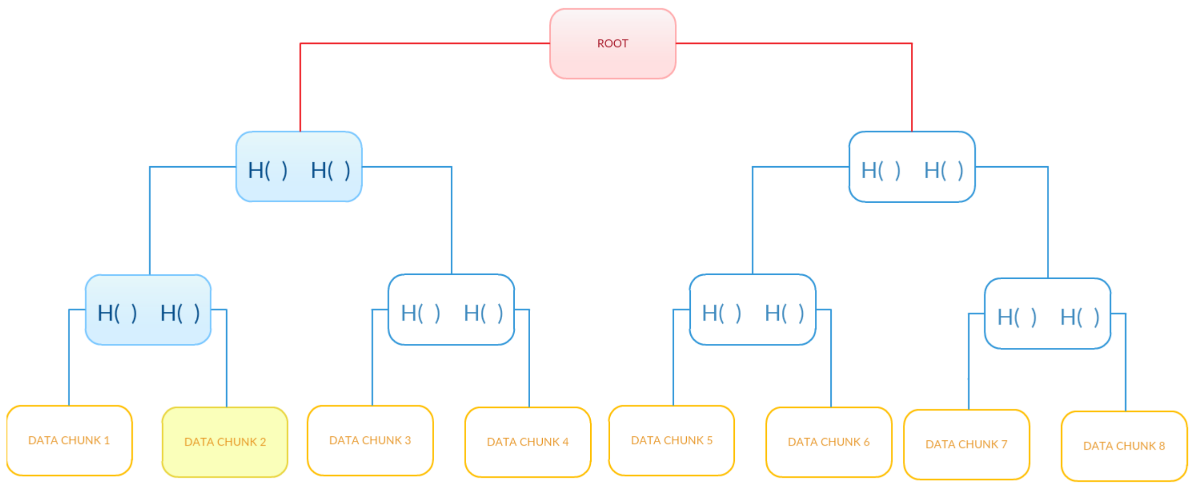


Figure 1.7: Block membership proof in a Merkle tree

1.4 Goofy-Coin and Scrooge-Coin

In this section, we will explore two simplified example of crypto currencies. These currencies are affected by shortcomings and drawbacks, but they are necessary to understand how to build a crypto currency from scratch. In this section, it is presented an incremental approach to satisfy all the requirements needed by a modern crypto currency. The first one is called Goofy Coin which is a crypto currency implementation naive like the Disney character. The second one is called Scrooge like the stingy Disney uncle Scrooge character because of the introduction of a centralised validator that can act avariciously.

1.4.1 Goofy-Coin

Goofy Coin is one of the most worthless crypto currency that can implement and it is used to explain useful concepts. It works respecting the following rules:

- Only one entity, Goofy in this case, can forge new coins
- Transactions are allowed among all users

This means that only Goofy can generate new coins that belong to him. It can build coins every time he wants and he is the central regulator of this currency. Coins are represented by a specific data structure where there is a unique ID to identify the coin and there is a valid signature of Goofy of that ID. In this way, everyone can verify that the coin is authentic. All coins can be passed to someone else. The transaction contains a statement that needs as input the two public keys of the two counterparts and a hash pointer of the coin. The coin owner will sign the transaction statement that says: "Pay the coin with this hash pointer to this entity named by its public key".

Each transaction contains a pointer to the previous transaction. Whoever now has that coin can prove the ownership of a valid coin showing the transaction list of the statements that started from Goofy, payed at the end the coin to him. Each user who owns a coin can pass it to someone else by signing a transaction statement and building the proper data structure. Anyone can verify the validity of this transaction scanning the chain of transactions and verifying the signatures along the list.

1.4.2 Double Spending Attack

The Goofy-Coin has an enormous architectural problem. Let's imagine that at a certain point, Scrooge receives a valid coin c_1 that can be verified through the chain of legitimate transactions starting from Goofy and that its hash pointer points to a valid coin data structure signed by Goofy. Scrooge wants to pass this coin c_1 to Huey signing a statement that says "Scrooge public key address pays c_1 to Huey public key address". Huey can present the chain of valid transactions starting from Goofy and ending with this last statement. Let's suppose that the

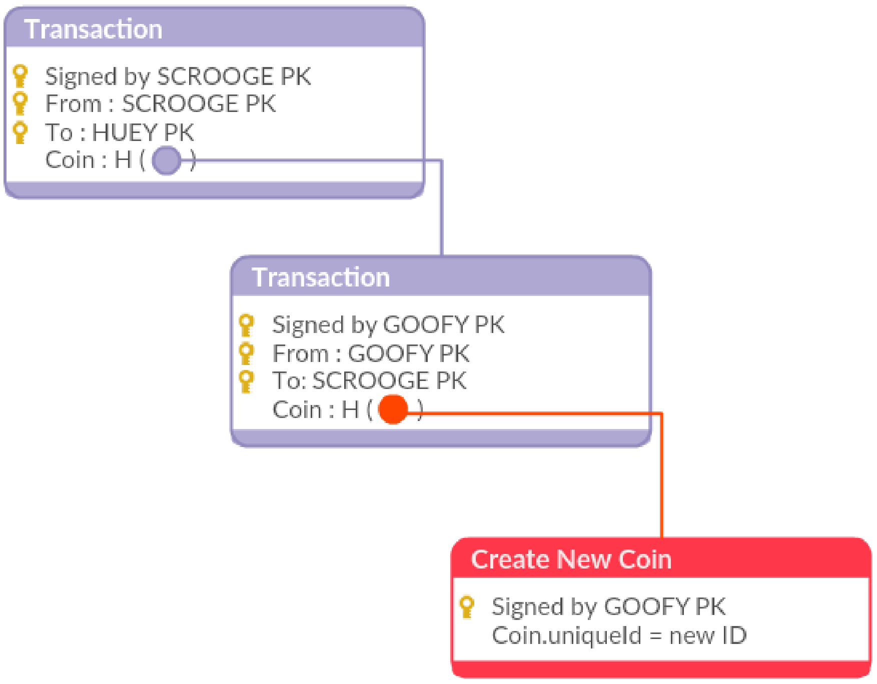


Figure 1.8: Transaction chain in Goofy-Coin

avid Scrooge passes the same coin c_1 to Dewey signing a statement: "Scrooge public key address pays c_1 to Dewey public key address."

Both Huey and Dewey could prove that their coin is valid even if Scrooge spent this coin twice. It is what is called a double spending attack. It can be even worse because imagine that the avid Scrooge can pass that coin more than two times giving it to Louie. Goofy-coin is vulnerable to double spending attack, therefore, it is not a safe currency.

1.4.3 Scrooge Coin

Since Goofy-Coin is vulnerable to double-spending , we will introduce a cryptocurrency that solves this problem. One of the simplest cryptocurrency that solves the double spending is Scrooge-Coin. The rules of Scrooge-Coin are as follows:

- Only Scrooge can forge new coins
- Transactions are allowed to all users.
- Valid transaction are stored in blocks, one per block, that have to been signed by Scrooge

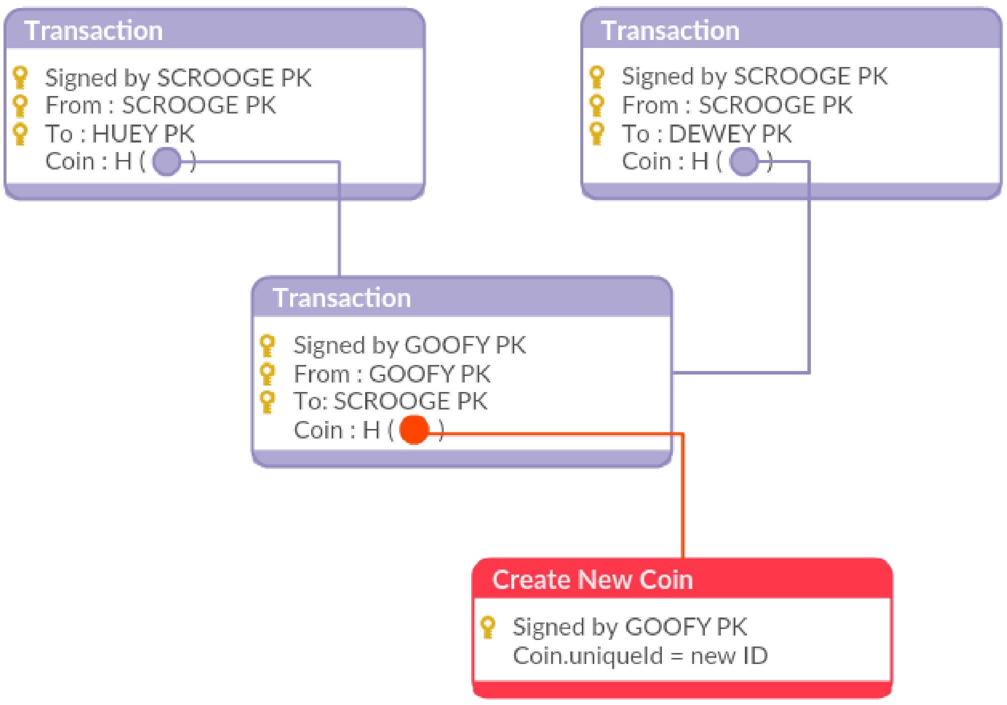


Figure 1.9: Double spending attack in Goofy-Coin

As for Goofy-Coin, Scrooge is the only entity that is allowed to do transactions that forge new coins with the difference that he can create more than one coins in one only one transaction, and they can be assigned to any address Scrooge wants. Each record in the transaction that creates coins represents a coin has an incremental value that distinguishes the coin within the transaction making it unique. Each record also has a value field that represents the coin's value in that currency and a public address which identifies the first owner of that specific coin.

The central entity Scrooge will take care to check the validity of each transaction, add it into a block and append the block to a blockchain of blocks. The blockchain will act as a history of the previous valid transactions. Scrooge will sign the entire data structure so that anyone can check its validity. For simplicity in Scrooge coin, each block contains only one transaction and a hash pointer to the previous block in the chain.

The transaction log will help to solve the double-spending. Let's suppose that this time Goofy tries to spend the same coin to Huey and later on to Dewey. Dewey will notice that Goofy no more owns that coin and he will ask for a valid coin. Scrooge will also not allow the last transaction from Goofy to Dewey, because that coin has been passed to Huey and will not

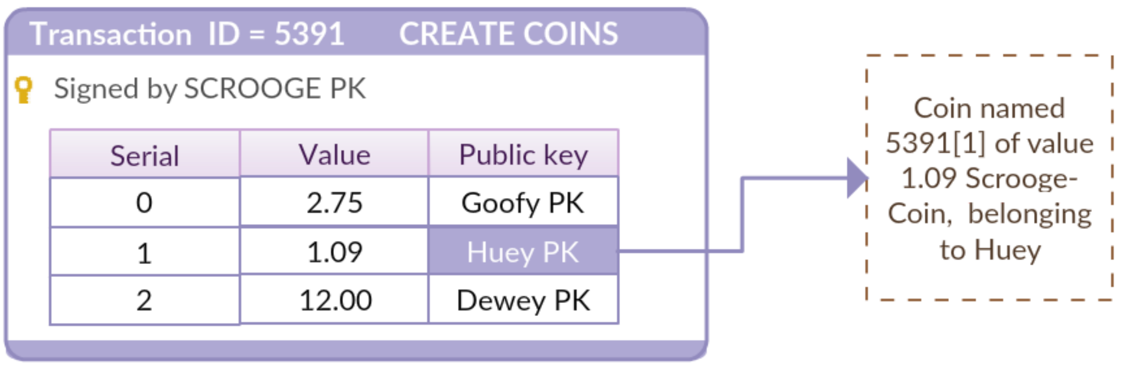

Transaction ID = 5391 CREATE COINS
Signed by SCROOGE PK
Serial
Value
Public key
0
2.75
Goofy PK
1
1.09
Huey PK
2
12.00
Dewey PK
Coin named 5391[1] of value 1.09 Scrooge-Coin, belonging to Huey

Figure 1.10: Create Coins Transaction in Scrooge Coin

appended the transaction to the Scrooge ledger.

Chapter 2

The Blockchain actions

From 2009 the year of the publciation of the Satoshi Nakamoto paper, Blockchain technology is emerging in different fields from criptocurrencies. List all of them will be impossible considering the vastity of the phenomenon and its continuous changing. However it is still possible to categorise some area of interests [PT17].

2.0.1 Cryptocurrency

Nowadays Bitcoin is the most popular cryptocurrency and many of the other popular cryptocurrencies derived from it (altcoins). There are thousands of cryptocurrencies but all of the as the following aspects in common:

- Decentralised ledgers
- Crytphographic tools (see last section)
- Principles of conceptual proof.

Chapter 3

The Message Layer

‘The limits of language mean the limits of my world’

***Ludwig Wittgenstein**, Tractatus Logico-Philosophicus (5.62)*

Bitcoin and other blockchain based projects were originally designed for cryptocurrencies and simple transfers.That is the reason why smart contracts and distributed scripts are not as powerful as today application’s demands. Moreover, transactions have scope only on their blockchain address space because of the simple original design. Consider that Bitcoin Script, the script language of Bitcoin takes inspirations by FORTH a 50th years old concatenative programming language. Many blockchains like Ethereum have implemented a complex business logic within the blockchain layer. Serpent and Solidity, the two Turing complete programming language that runs on Ethereum are inspired respectively by Python and Javascript two of the most adopted scripting language today. As we mentioned blockchain, protocols cannot accept changes without fork their chain into two different ones. What will happen in 50 years when both Python and Javascript will be old and obsolete? This is the reason why we think that the control logic as well as the business logic, should be decoupled from the transaction level.

3.1 Standardization

This layer abstracts all the transactions in the same space regardless the particular ledger. Applications then design the rules to choose, which information are useful in their logic and in which order they need to be sorted. This layer extracts all the information about the transactions from every single ledger. This layer stores transaction information related to the ledger, the source, the recipient, the amount of coin, the script used, the smart contract and everything the transaction can store in a particular ledger. It allows legacy applications to work with our architecture and extend them their portfolio capability regarding ledgers and functionality. However, ledger technologies are very different from each other, and this requires building a standard naming system of the message layer. The standard should define how to refer to the different ledgers and to their particular fields in a unique way. For example, it should be possible to refer to the address of a Bitcoin transaction using a compound syntax (e.g. **BTC**.Height.Transaction_Index.Output_index should apply to the output at index "Output_index" of the transaction at index "Transaction_Index" in the block of the main chain with the height equals to "Height" parameter.) The standard will give the opportunity to build standard libraries for developers to interact in similar ways with the different ledgers and boost this architecture.

3.2 Messages and information out of the chain

Many blockchains offer the possibility to add messages in the transactions. Bitcoin has the OP-RETURN field, Ethereum has the field that is used for the script bytecode, Ripple has the 'memos' field. External messages can be used for an arbitrary logic. These messages can be considered transactions or block of an external blockchain or locally stored by users. Only a fingerprints of these messages are added on the blockchain. The Digital Asset Platform use this approach for privacy constraints and to solve the reconciliation problem. Each user has its own set of view and can build its view of the transactions. Only the hashes of these transactions are stored in the ledger to prove their validity. In case of conflict between two views, a user can show the validity of its set of transaction showing the transactions that contain the hash of his messages and prove their validity to the counterpart. We want to extend this approach to messages. Messages are an exchange of information; they can be a simple transaction or something more complex concerning the application logic.

In the figure 3.1 there are different applications that use this approach. The first white block represents a section of a blockchain where the applications are appending messages. The mes-

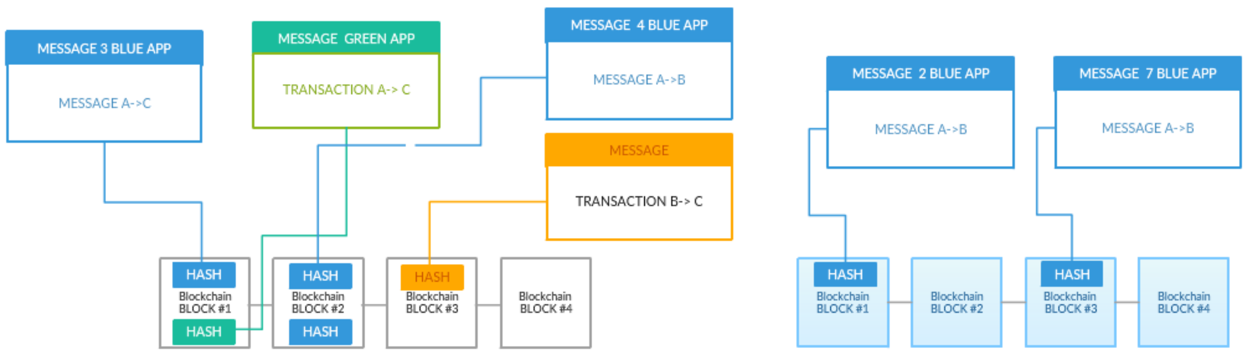


Figure 3.1: Messages out of the chain of different applications

sages of the different applications are differentiated using different colors. The blockchain hosts only the hash of those messages. Each hash corresponds to a message of one of the applications. There can be more messages of the same application in different transactions in the same block (block n.2 in the white blockchain) and there can be messages of different applications in the same block (block n.1 of the white blockchain). One application can run on different blockchains, in this case, the blue application has some messages on the white blockchain and some others on the blue one. Notice that the messages' number is monotonically increasing.

3.2.1 Message Format

The message is a more generic concept than the transaction, in fact, they can require a more complex format and be part of more complicated interactions between the parts. Hence, markup languages like XML or JSON can be a good fit to build messages in this context. The application could define the valid sequences of messages and their format through a schema to validate them. Moreover, these messages can be exchanged over standard HTTP machines and XML, and JSON a good choices since they are already widely used in Internet communication.

3.3 Messages and security Properties

3.3.1 Identification

Identification is the action to claim to be a certain entity, e.g. a person, a company, a department with no ambiguity. In login system, a user identifies himself through the username, usually an arbitrary alphanumeric string. In blockchains, the identification is usually achieved through public keys or their hash. In this context public key seems to be the best approach but there can be cases where they can also be used the username. The message format should always have a field for the senders and the receivers.

3.3.2 Authentication, Authorization

Authentication is the action to prove that someone is who claim they are. In standard login systems, a user can authenticate through a password. In the blockchain, authentication is done through a private key in the process of the digital signature. In this context to consider a message compliant with a user will and to be sure that users involved are really who they claim to be, asymmetric cryptography and digital signature are the accepted approach. However it is also possible that the application after the initial authentication phase with username-password shares a secret with the user and then starts to exchange messages encrypted with symmetric cryptography. The hash of these messages are stored in the blockchain, and the user with the shared secret and the complete message would be able in the future to prove that he received those messages.

3.3.3 Non-repudiation

Non-repudiation is the action to do not disown an action happened in the past. Digital signatures are the common way to achieve this property in computer science. In this context, we have the non-repudiation property on the message content and his order among other messages. It means that if someone has a copy of the signed message they can state that the message was compliant with the will of the entities that sign it.

3.3.4 Privacy constraints

Many applications need to hide to users data that are related to other users to be compliant with privacy requirements. The classic lower level of blockchain cannot satisfy privacy constraints because of the transparency property of the blockchain.In other words, in the common scenario the transaction cannot be completely anonymous or it would be not possible check their validity. The Digital Asset Platform uses a variant of the regular blockchain where only the hash of transactions is added in the chain. We can use the same approach adding the hash of the messages in a special field of the regular blockchain's transaction. Coloured Coin uses a similar mechanism using the OP_RETURN field of Bitcoin. This solution can have some drawbacks because everyone can add the hash of a message into blockchain transactions. It means that there can be invalid messages whose hash is the chain. Therefore, the presence of the hash in the chain does not mean that the message is valid. Please consider the strategies discussed into 3.3.2 to understand how to check the message validity. If someone has a copy of a valid message (e.g. signed by all the stakeholders), he has the opportunity to show its hash is in the chain, hence its validity.

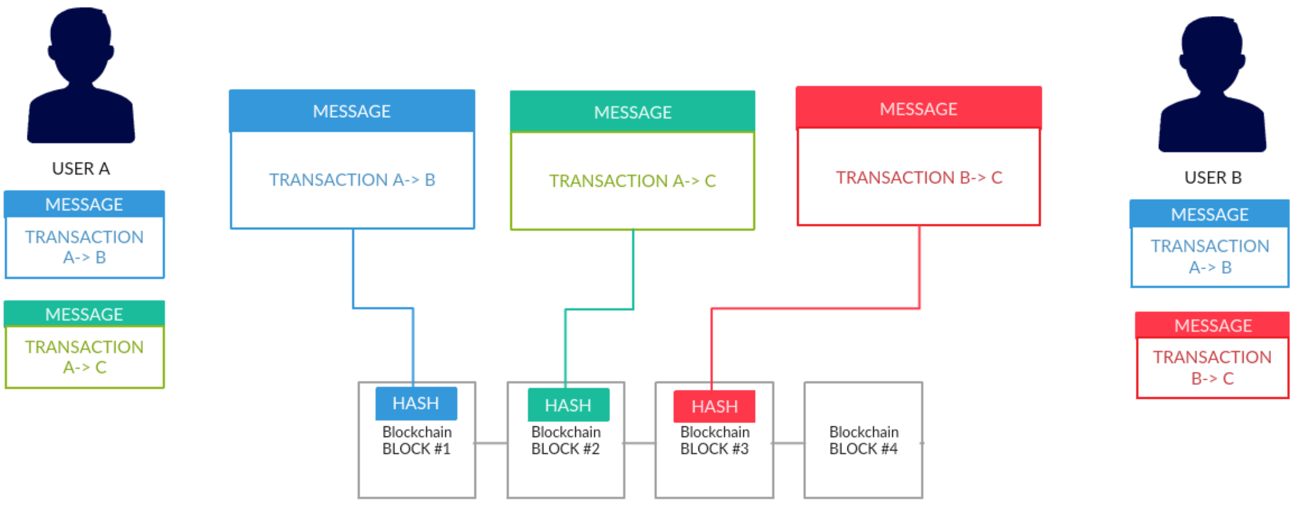


Figure 3.2: Messages out of the chain and user's views

In the figure 3.2 there are messages exchanged out of the chain for example sending them through the Internet. Their hashes are stored in a different block of the chain even if it is enough to save them in separate transactions. User A can see all the hashes in the chain but has details on his own transactions: the blue one and the green one. He does not know that there is a transaction between the User B and the User C. If there are conflicts between user's view, each user can show the complete transactions and prove their validity by sending the hash pointer in the blockchain.

3.4 Improve Privacy specification

3.4.1 Multipart message

In the section 3.3.4 we show a way to give the proper view of the application to the proper entity. In some context, applications can require messages to have a hierarchy that has different scope regarding privacy. Since we have arbitrary message format, we can introduce hash pointers to another part of the message. In this way, the application can provide each user with the proper parts of the messages. If the user has no right to see some part of the multipart message he will only see a hash pointer or a random piece of data.

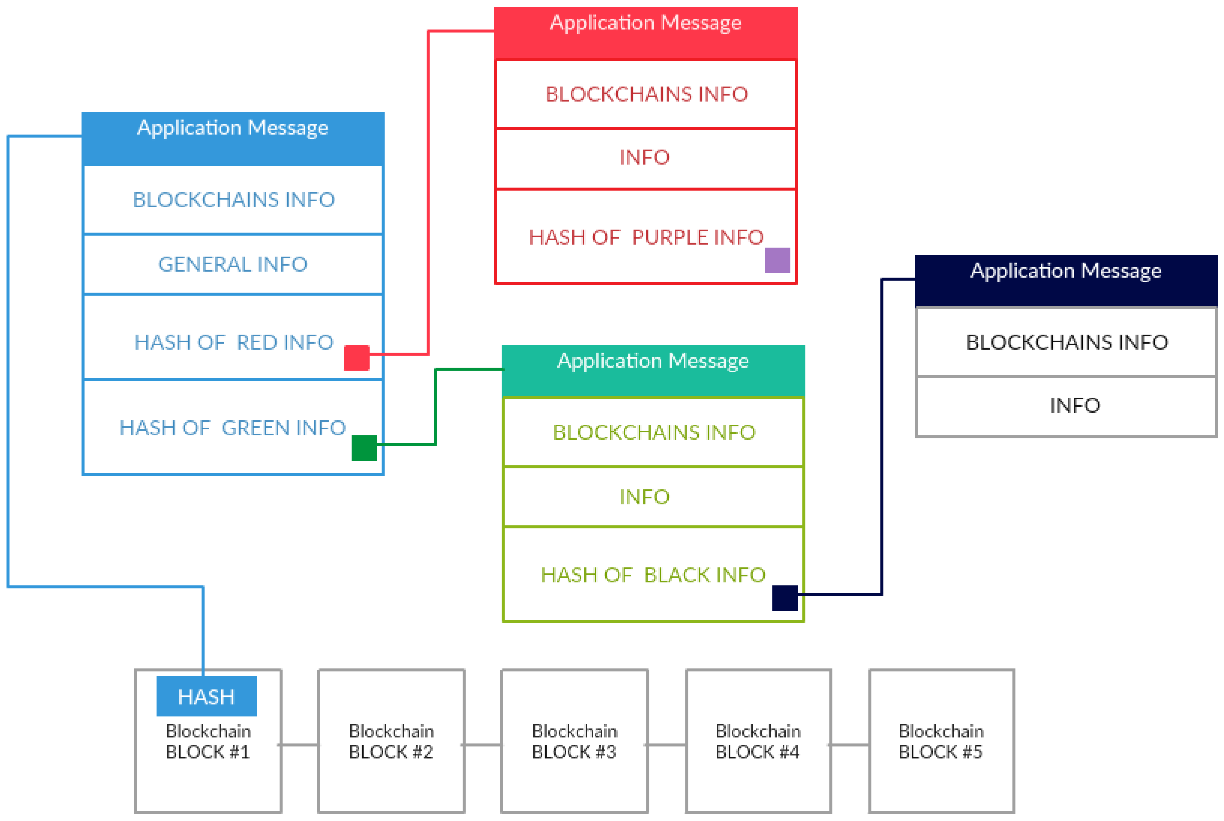


Figure 3.3: Messages out of the chain and different level of secrecy

In the figure 3.3 the user that has the right to see only the blue part knows only that there are two other branches of the message, the one that starts with the red part and the one that starts with the green one. However, he does not know how deep these branches are. The user that can see the black part knows about the green one and the blue one, but we cannot know if he is aware of the content of the red part. Please consider that this modular approach can be replaced by removing the pointers from the lower part, adding a reference to the upper ones and treating them as newer messages adding their hash on the chain. However, this can lead the business logic to inconsistency if the other part was not correctly added or something happened in the between.

3.4.2 Hide level of secrecy

The problem with the approach in 3.4.1 is that who sits on a level of privacy knows if there are other levels of privacy. This is something that should be avoided in some context. We can take the fields that contain private information like hash pointers encrypt them with the public keys of the interested user and make it look like a random piece of data to other users. This means that in the case that there are no more levels of secrecy we should add a random sequence of data.

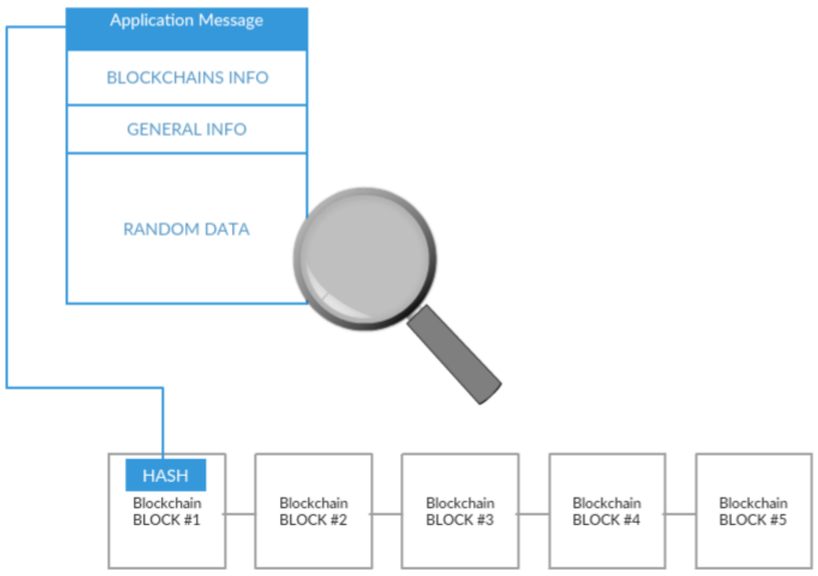


Figure 3.4: The field to extend the message contains the hash pointers of the other parts of the message also contains a random sequence of bytes

One of the problems with this approach is that the secret information can be different in size and it means that we need to fix this size and use a padding strategy. The biggest problem is that we need to make different people aware of the different part of the message. Since a message can be encrypted only with one key if we want to use this strategy we need to replicate the different parts concerning the particular key. Let's consider 3.5 where we have three users that can see the green part and of these three only two the black one. We need to add three green hash pointer. One of these three, let's say the first one has as extension field random piece of data, the second a hash pointer to a black message encrypted with its own key, i.e. the second one and the last one with the third key. The number of the keys involved is given by the number of lowest level.

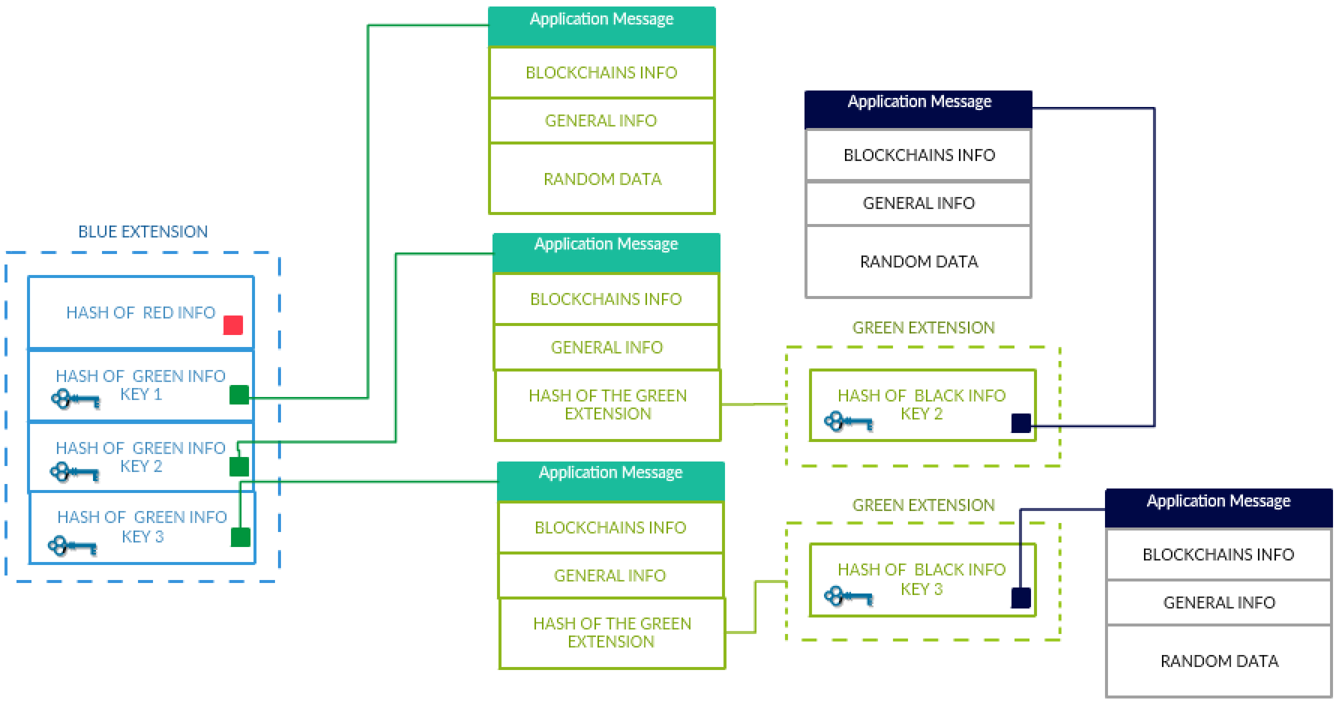


Figure 3.5: Cryptography with multi-part messages

3.4.3 Extension Field

Even if we showed with 3.5 that is still possible to manage multi-part messages with different keys we still have the problem with different sizes with the need to fix this size to make not authorized people aware about how many branches of secrecy there are at that level. In this section, we propose an approach (3.6) that partially solves that problem by adding only one field per message that can point to a message part that only contains hash pointers. IN this case, the blue message has only one extra field for the extension. The content of that field can be a random sequence of bytes or the hash of the blue dashed extension. The blue extension contains the pointers to the red part and the green part. In this way, people who sit at the lower level do not know if there are other levels of secrecy and how many they are. However with this solution the people that can read the red part of the message know about the green one, as well as who can read the green part know about the red one.

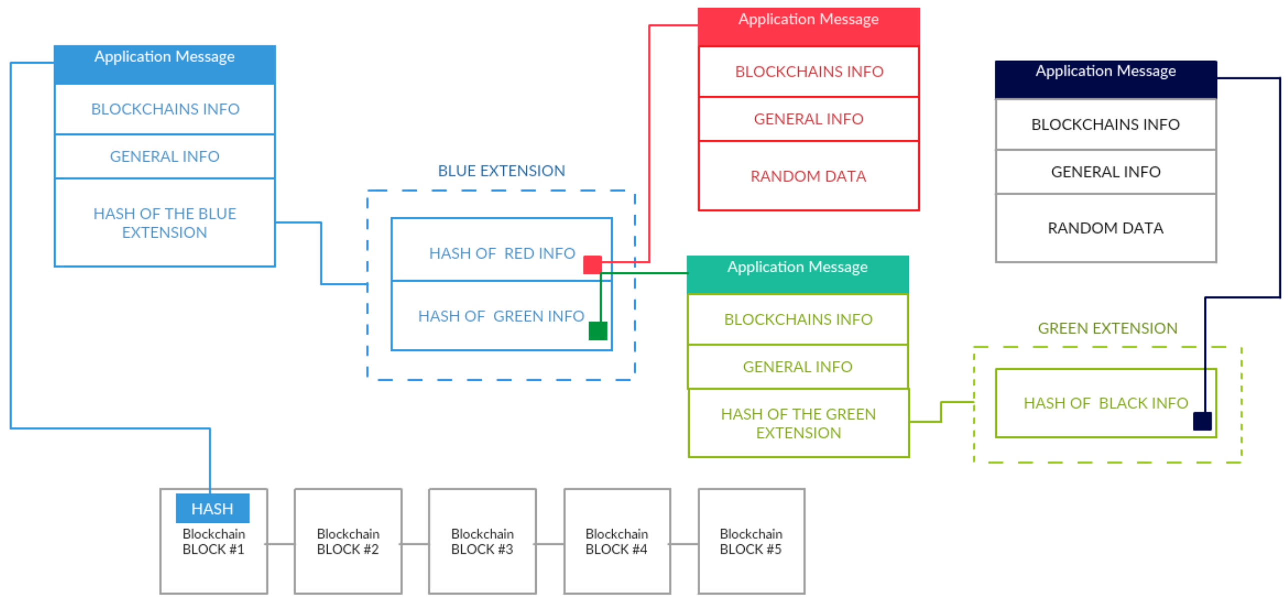


Figure 3.6: Level of secrecy without keys

3.5 Shortcomings of Messages out of the chain

The issue with building an over-layer of the message whose hash is added on the blockchain is that we cannot control who and which messages can be appended. It means that if there are no rules to limit it there can problems like trashing and spamming. It is also possible to design a denial of service. It is what would happen in the lower level if the blockchain has no consensus mechanism. In fact, the consensus rules how frequently blocks are added in the chain.

3.5.1 Spamming

In a messaging system, spamming is the action to send unsolicited messages. This action results into the overwhelming of the channel with valid messages. In this case, the channel is the space per transaction of the field we use to add the hash of the application message. One possible solution can be to let only a third party validate messages by signing them. In this case, only messages that have the right format, that have a sign of the message by the authority and that has a hash in a valid chain can be considered valid. This makes our application centralized, however, in some context this can be mandatory, and it still gives to the user the transparency and non-repudiation property that traditional centralized application cannot guarantee without trust. Without a centralized authority that checks the message consistency and controls its growth, we can also occur in replica issues or replay attack. Consider a message that is valid, and it is added many times if that message contains a command it could be executed an arbitrary number of times. Also, it becomes more difficult to build the application logic if during the scanning of all messages there are messages that have the correct format and have the right signatures but are no longer valid because they are already in the over-ledger. There are techniques to solve this issues, like enumerating messages or using timestamps.

3.5.2 Trashing

Trashing is the action to overwhelm a channel with meaningless data regarding the business logic (not-valid messages). It can be the action of a malicious user or as it can happen in this context also be an involuntary act. In fact, notice that many application can run on the same blockchain. Messages of one application are trash data for the other application. It means that if an entity of the application loses the pointer to the particular block transaction that hosts the hash of the message, he should compare this hash with all the hashes of the valid messages of all the application that have run into that blockchain until finding the matching one. It also means that to check if a message has been validated and added to the chain, users have to check all the messages and then store the pointer to it.

3.6 Speed and throughput of messages

Building new applications on the blockchain layer affect the speed parameters concerning the underlying layers. If we want to act on the over-layer, we must accept the latency of all the underlying technology involved. In the trivial case if we want to perform an over the action that affects only one blockchain we want to be sure that its hash is stored in a valid transaction. In some technology like Bitcoin to be sure of the commitment of the transaction we usually need to wait for six more blocks to reduce the probability that someone can come up with a longer chain that making our transaction invalid. Now let's suppose we have our action ready at time t, we need to wait for a latency that is not deterministic before our transaction is accepted and added in a block. After this period we also need to wait a probabilistic time until the six next block were appended. Now if we have operations that require many operations on a single blockchain we need to consider this time for each succeeding transaction all this amount of time.

Chapter 4

The Filtering and Order Layer

'Ordo et connexio idearum idem est ac ordo et connexio rerum'

***Baruch Spinoza**, Ethics (Part II, Prop. VII)*

One of the blockchain's benefit is to store the history of blocks in a way that can quickly detect any changes of the block's order or manipulation of data within the block. A consensus mechanism embedded in the blockchain 's protocol decides the order of valid blocks unequivocally. Filtering and sorting blocks allow us to trace the ownership of an asset or calculate the balance of an account by replaying all the transfers and transactions from the beginning of the chain. These transactions and transfers are secure because of digital signatures that give the blockchain the identification, authorization and non-repudiation properties. The signature algorithm, the key length of public and private key pair and the hash function to produce the account address from the public key are embedded into the particular blockchain protocol. The order property and the not-repudiation property together solve the double spending problem within the blockchain 1.4.2. The challenge of building an over ledger upon many ledgers is to find a criterion order to choose among different blocks of different blockchains which block comes first and how to translate the address format of a blockchain in a way which is compatible across other blockchains. In our reference architecture, all the information contained in all the ledgers are put together in the a logical (not physical) layer where a set of rules filter a small set among them and decide their order without regard to their position in their own ledger.For each transaction, we can extract information regarding the sources, the receivers, the coins, the script, the contract and combine them to define rules. Some ledgers allow users to append arbitrary data to transactions. User can use this space to fill these fields with messages or if it not enough space with their fingerprints. Even if these messages are meaningful in the hosting transactions we can use the contained information to filter and order messages and transactions in different ways. Consider that it is straightforward to find an order among transactions in the same ledgers it can be challenging to do it in a multi-ledger environment. In the previous section we focused on how to build this message and what we can achieve with them. In this section we will introduce the ordering problem. In this section, we will focus on

- How to set an order binary relation for blocks of different blockchains
- How to allow cross-ledger transactions by mapping addresses of the various blockchains into one address valid in the over ledger domain

4.1 Filtering Criterion for the over ledger

The message layer contains potentially infinite and messages and information which are all the possible combinations of the information that exist in all the ledgers. Find a way to bound this set is to introduce filtering rules to drop all the messages we are not considering in our overlegder. All these rules are related to the transactions and their fields. However some of the fields can be filled with arbitrary data that we used for the out of the chain messages. This means we can set rules on these messages because their fingerprints are in the ledgers. We have two levels where to apply these rules.

- Transaction validation rules
- Out of the chain messages validation rules

The first category can be a set of rules that uses information on the transaction. This set of rules includes the choice of allowed ledgers to produce messages. Once we chose the ledgers, we need to decide which fields select. This is hard because each ledger has its transaction structure, but we need to select fields to build consistent messages among the ledgers.Even if we chose the set of shared fields it is difficult to create messages among the ledgers which are compliant. For example, if we select two ledgers which have a similar transaction structure and we only look to the addresses (source and receiver) we will need to find a way to map them to other to allow message exchange among ledgers. For this reason out of the message are useful because they give us the ability to implement this map and eventually more complicated information. In fact, we can map two address in two different transaction ledgers external messages. These messages can contain a signature of a user and their fingerprints in the transactions. We can also apply complex logic to this messages utilising validation schema for example. However, transaction information is essential because they can contain low-level information about addresses and coin information. For example, an application can allow some messages only if they pay to one of its low level (transaction ledger) addresses a certain amount of coin and then use the source of that transaction to send the response. Eventually, we can also use script-smart contract information to allow a legacy application to work.

4.2 Ordering Criterion for the over ledger

4.2.1 Order binary relations

The branch of mathematics that studies the order using binary relations is the order theory. It provides formal tools to compare two objects within a set and decide which one is greater than the other. Consider that there are objects that are comparable in many different ways. It is possible to sort the set of natural numbers with their magnitude or with the lexicographical order. For the complex object, composed of various fields, we can compare them using the combination of their fields.

Partially ordered set

Given a set P and a relation $\leq$ on P elements then $\leq$ is a partial order if and only if for all x,y,z in P we can ensure these three properties:

- Reflexivity: $x \leq x$
- Antisymmetry: if $x \leq y$ and $y \leq x$ then $x=y$
- Transitivity: if $x \leq y$ and $y \leq z$ then $x \leq z$

These orders are also knows as poset. If we can prove only reflexivity and transitivity we can call the order a preorder. If the relation has the antisymmetry and transitivity property and instead of the reflexivity property has the irreflexivity property:

- Irreflexivity: not x<x

we define this relation a strict partial order relation. If we can prove the previous three properties and the following:

- Totality: x≤ y or y≤ x

We can define the relationship as a total order and the set of a total ordered set or linear orders or chains. In Poset, there can exist unique elements like the least element l≤ x, for all x in the set and the greatest element x≤ g for all x in the set. In partial order, there can be items that have no elements greater/less than them and that are not comparable to each other. In this case, these elements are called respectively minimal and maximal. There are elements in the set that are special concerning a subset of the order. Given a poset P, a binary relation ≤ and one of its subset S, a lower bound of S is an element l such that it is less or equal than all element s in S. Conversely with the same assumptions an upper bound is an element u such that all elements s in S are less or equal than u. Another important concept is the infimum and the supremum. The infimum is a lower bound called inf of a subset S of a poset P in which is defined the relation ≤ if for all lower bounds l of S in P l≤ inf. Hence the supremum is an upper bound called sup of subset S of a poset P in which is defined the relation ≤ , if for all upper bounds u of S in P, sup≤ u. If the infimum is contained into the subset S, the infimum is called Least element, in the same way, if the supremum is contained in the subset S, it is called the Greatest element. If an ordered set S has the property that every subset of S that is non-empty and it has an upper bound, it also has the least upper bound, then it has the least-upper-bound property also known as Dedekind completeness. If a total order has a least element for all its subsets that are not empty, the order is called well-order.

Zorn's Lemma

An important proposition of the set theory is the Zorn's Lemma that states: Given a poset P with the property that every chain contained in P has an upper bound in P; then the set P has at least one maximal element.

Duality

The previous definitions of the least element or the minimal set can be obtained by the greatest element or the maximal by inverting the ordering function. This operation allows building new order starting from an order.

4.2.2 Representation of a Poset

Hasse Diagram

One way to represent poset visually is Hasse diagram. Hasse diagrams are graphs where the elements of the set are vertices. A binary relation is represented by an edge between two vertices that represent two elements such that the vertex that is below is the predecessor of the relation and the one which is above is the successor of the other vertex. Orders are bottom-up represented. In the figure 4.1 it showed the Hasse diagram of the poset P(*,S) where S is 1,2,3,5,6,10,15,30 the divisors of 30 and is divisibility relation that exists if one element is an integer divisor of another one. For example 1530, because 30 divided by 15 has no reminder, therefore, there is an edge that connects these two elements, and 15 is below 30 because the binary relation is 15*30. The topmost element is 30 because in the set there are no elements for which 30 is a divisor. This is a way to visualize the existence of maximals. It is not possible to spot the existence of greatest elements it is possible that there is a vertex that has a predecessor and that it is not connected to the presumed greatest element or any of its predecessor. In the same way with this representation is easy to spot minimals but not that easy to understand if there is the least element.

Directed Acyclic Graph

Another way to represent a poset is DAG (directed acyclic graph). Each vertex represents an element, and the edges point from the predecessor to the successor.Dags are data structures easier to implement, and they can be used to check properties or find particular elements. This

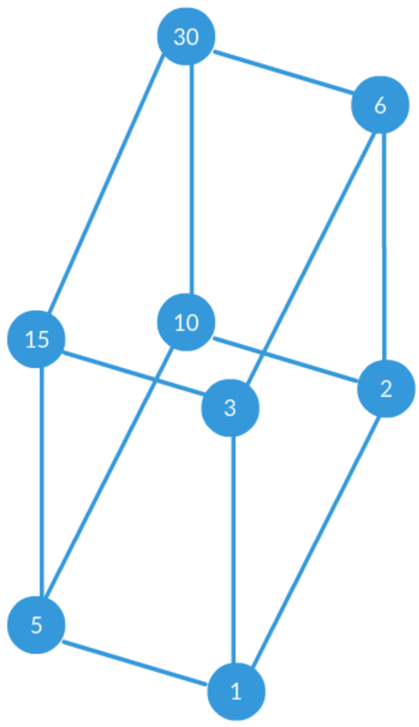


Figure 4.1: Hasse Diagram of all divisors of 30, ordered by divisibility

representation offers a way to find the set of minimals and maximlas in O(n), where n is the number of vertexes, by checking node by node if the list of outgoing edges is empty (maximals) or if the list of ingoing edges is empty (minimal). If one of these sets has only one element, it is a sufficient way to prove the existence of the greatest element or the least element. It is also possible to check if an element is a predecessor or a successor of another one by checking if there exists a path that connects the two elements in O(V+E) with a Depth First Search (DFS).

4.2.3 Blocks order in popular Blockchains

The consensus is the way blockchain can decide which block will be added to the ledger. Blocks are added one, by one and among block contenders, only the ones that respect all the consensus rules are eligible to be added to the chain. These rules check the consistency and validity of the block content and to prevent block's spam. Bitcoin and Ethereum use a consensus mechanisms based on mining called respectively HashCash and Etash. Mining is based on Pow, i.e. the problem to produce a string that is easy for other nodes to verify and hard to build. For a block to be added to the ledger, miners must solve a PoW problem that uses as input a digest of the

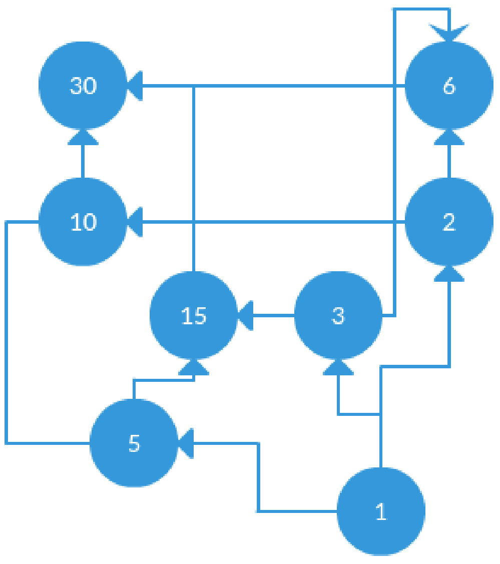


Figure 4.2: DAG of all divisors of 30, ordered by divisibility

current block and a digest of the previous one. In this way it is easy to build a strict binary relation among blocks. Nodes are Incentivised to solve PoW because they add a newer block get redeem a reward, tipically in the format of the cryptocurrency. Ripple instead uses an iterative process. Each node builds a transaction set based on the newer transactions it knows and broadcasts it to its trusted network as a proposed transaction set to be applied to the ledger. When a node receives a proposed transaction set by trusted nodes, it implicitly votes each transaction into the set with yes or no by including it in a newer proposed transaction set. If a certain percentage of the nodes in the trusted network of a specific node votes for a transaction then that node will add it to its transaction set. After a while, the percentage threshold rises making the transaction set converging into one that is added to the ledger concerning a specific timing scheme. Another technique to reach consensus is through PoS (Proof of Stake) where the creator of the new block is selected in a deterministic way depending on its stake. Ethereum is probably going to hard-fork turning into a PoS consensus system rather than a PoW one.

4.2.4 Blocks and transaction in the over ledger

All the consensus techniques allow the choice an order among the blocks thus it is trivial to represent with a Hasse diagram the order of blocks B(<,S), where S is the set of all blocks and the < relation states which block comes firs. B is a total strict order it means that given two blocks it is always possible to choose which one comes first.

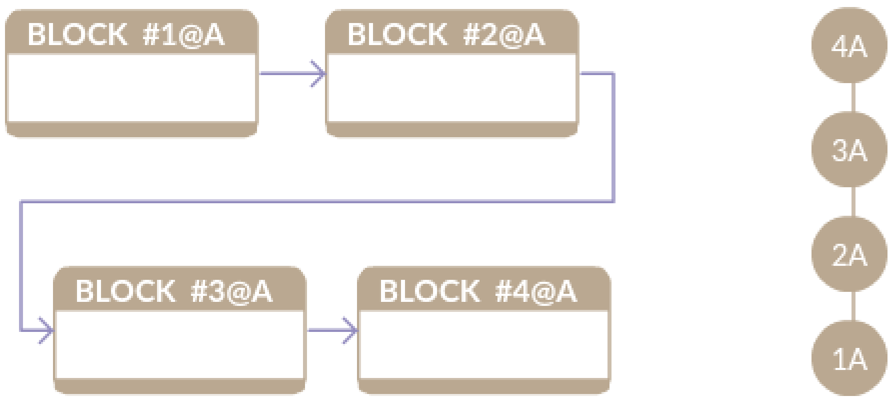


Figure 4.3: Hasse diagram of the order B(<,S), where S is the set of blocks {1A, 2A, 3A, 4A}

If two blocks have two conflicting transactions, the transaction in the greatest block is not valid because that coin is considered already spent. This is the way that ordering partially solves double-spending. One problem is what happens if the two conflicting transactions are present in the same block. Since transactions within the same block happen at the same time, we can model the set of transactions as a total poset T(≤,W) where W is the set of all the transactions contained in the blocks and ≤ is the binary relation that compares transaction order.

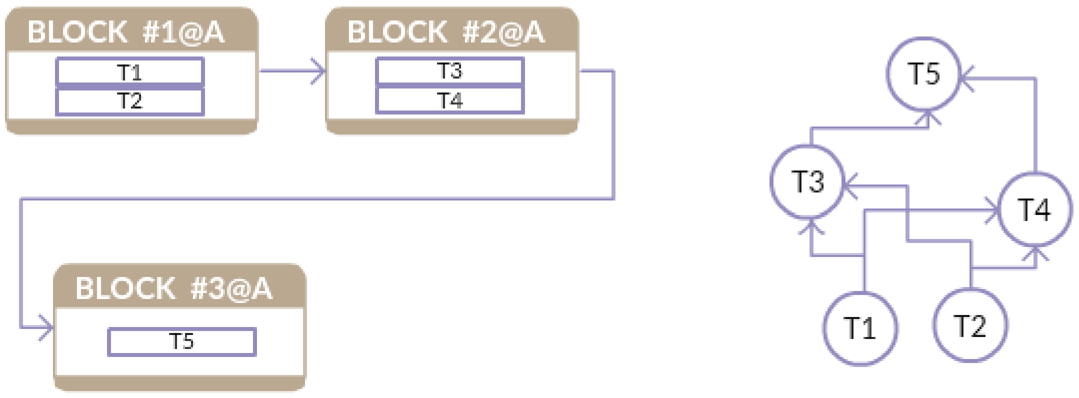


Figure 4.4: Hasse diagram of the poset T(<,W), where W is the set of transaction {T1, T2, T3, T4, T5}

In many blockchain protocols, Bitcoin included, blocks with the conflicting or invalid transaction are not eligible to be added, and when they are proposed as the next block in the ledger, they are dropped. In this way, it is not possible to have conflicting transactions either in different blocks or the same block.

4.2.5 Over ledger block's order

While the order set of blocks and transactions within a particular blockchain can be a model as total order, comparing the of blocks or transactions in different blockchains makes not possible to compare different element that exists inside a different blockchain domain.

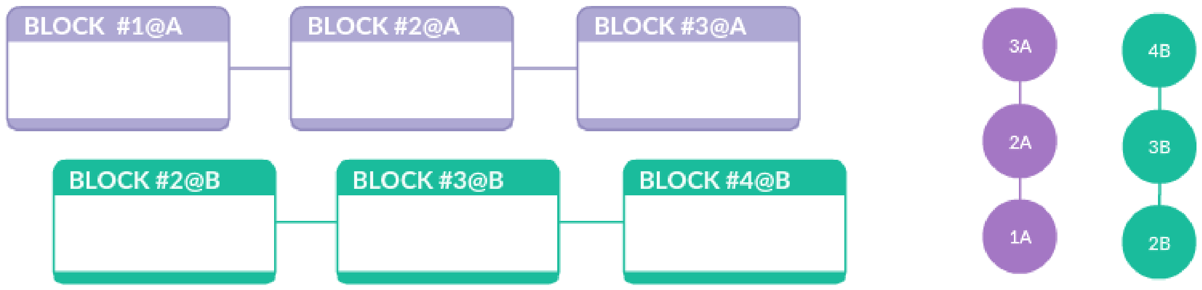
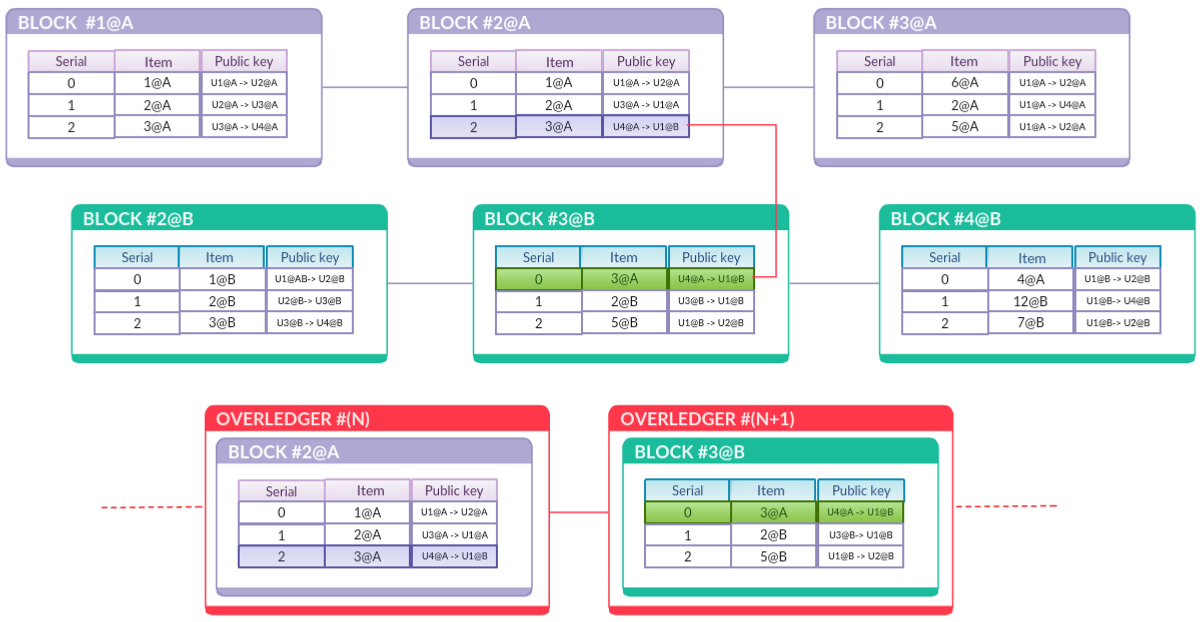


Figure 4.5: Blockchains and over ledger representation

The 4.5 shows the Hasse diagram of the set of some blocks of the two different blockchains A and B. The represented order is the Poset O1(≤,S) where S is the set { 1A, 2A, 3A, 2B, 3B, 4B } and ≤ is the binary relation that decides which block comes first. There is no way to compare the block 1A to the 2B or any other couple of blocks belonging to different chains. The easiest idea is to use timestamp to decide which one comes first, however the different consensus mechanism, therefore, the timing scheme that regulates the way new block is added can make this choice not compliant with the inter ledger business logic, moreover it is possible that some chains do not use a timestamp that is comparable with others. However, this is not a problem in overledger building because there is no need to compare transactions that move assets that exist only in their blockchain domain and it is not possible to have conflicting transactions. The situation changes when we introduce cross ledger transactions, in this case, we can compare some of the blocks in different chains and detect conflicting transactions.

4.2.6 Cross Ledger Transaction and block poset

In the 4.6 there are two blockchains A (the purple one) and B (the green one). Each block contains a set of transactions identified by the unique field serial and composed by a 'Coin' field that indicates the asset moved in the transaction and a 'Public key' field that shows which user is moving to whom the asset. In the block 2 of the blockchain A (#2@A) the transaction indexed 2 is a cross ledger transaction that moves the asset 3 of the blockchain A (3@A) from the user U4 in the blockchain A (U4@A) to the user U1 of the blockchain B (U1@B).



BLOCK #1@A

Serial	Item	Public key
0	1@A	U1@A -> U2@A
1	2@A	U2@A -> U3@A
2	3@A	U3@A -> U4@A

BLOCK #2@A

Serial	Item	Public key
0	1@A	U1@A -> U2@A
1	2@A	U3@A -> U1@A
2	3@A	U4@A -> U1@B

BLOCK #3@A

Serial	Item	Public key
0	6@A	U1@A -> U2@A
1	2@A	U1@A -> U4@A
2	5@A	U1@A -> U2@A

BLOCK #2@B

Serial	Item	Public key
0	1@B	U1@AB-> U2@B
1	2@B	U2@B-> U3@B
2	3@B	U3@B -> U4@B

BLOCK #3@B

Serial	Item	Public key
0	3@A	U4@A -> U1@B
1	2@B	U3@B -> U1@B
2	5@B	U1@B -> U2@B

BLOCK #4@B

Serial	Item	Public key
0	4@A	U1@B -> U2@B
1	12@B	U1@B-> U4@B
2	7@B	U1@B-> U2@B

OVERLEDGER #(N)

BLOCK #2@A

Serial	Item	Public key
0	1@A	U1@A -> U2@A
1	2@A	U3@A -> U1@A
2	3@A	U4@A -> U1@B

OVERLEDGER #(N+1)

BLOCK #3@B

Serial	Item	Public key
0	3@A	U4@A -> U1@B
1	2@B	U3@B-> U1@B
2	5@B	U1@B -> U2@B

Figure 4.6: Cross-ledger and Over-ledger transaction representation

The same transaction appears according to the blockchain B semantics into the block $\#3@B$ at index 0. For simplicity, we assume that the blockchain semantics are the same. These two transactions represent the same transfer, therefore, it is evident that the two blocks that contain these two transactions happen together concerning our business logic. However, since these two blocks can be added with different consensus method it is very unlikely that they are added at the same time. Moreover, it is mandatory to find a way to make atomic the cross ledger transaction, in the case that only one of the two blockchains recorded it, generating inconsistency. In this over-ledger representation even if the transactions of these two blocks happen at the same time regarding our business logic they are added into following over blocks. This choice will be explained later when introducing a protocol to achieve atomicity the user where the asset belongs will first propose the transfer.

In 4.7 there is the Hasse diagram of the set of all the blocks in the different blockchains ordered by the binary relation that states which block comes first in the over ledger business logic. While in 4.5 it was not possible to compare blocks in different blockchains in this case because of the cross ledger transaction we can say that $\#1A \leq \#4B$ or $\#2B \leq \#3A$. Thus, we can model the over ledger as a chain that contains blocks of the different chains. A block of the

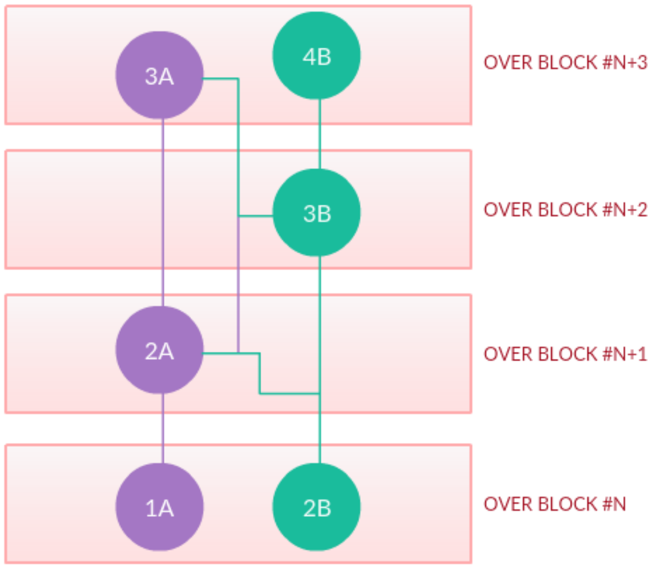


Figure 4.7: Hasse diagram of Poset O(≤, S) where S are the blocks in the blockchains in 4.6

over ledger can include one or more block of more blockchains. This poset has for every block a set of minimal blocks because of Zorn's lemma 4.2.1 and duality 4.2.1. In 4.8 is represented the overledger based on the cross ledger transactions shown in 4.7.

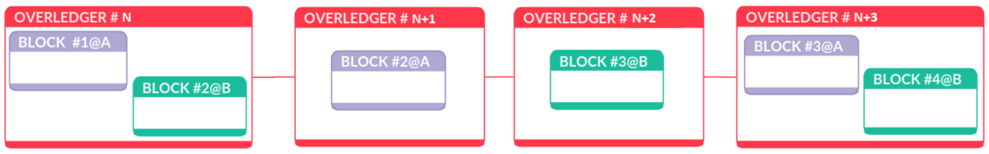
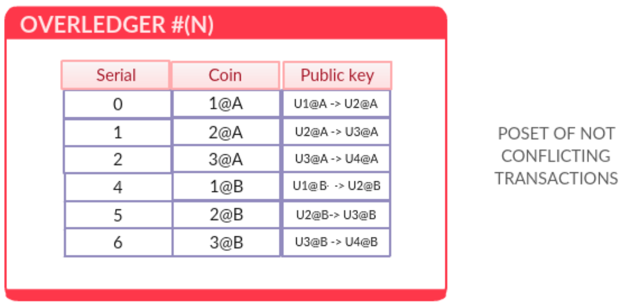


Figure 4.8: Overledger modelled as a blockchain of block set 4.6. Note that the block 2 and 3 can be grouped together

Each block of the over-ledger contains many blocks and therefore the set of all transactions into those blocks. In the example shown in 4.6 we can model the over block as a regular block (4.8 with all the transactions because we are sure there are no conflicts, therefore, we can surmise that they happen in the same instant.

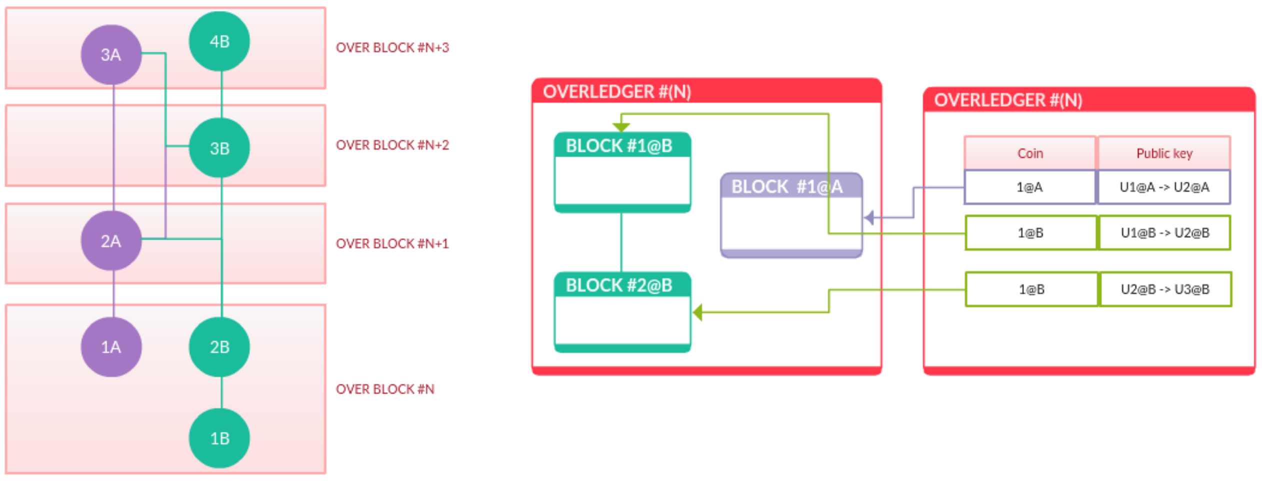

OVERLEDGER #(N)
POSET OF NOT CONFLICTING TRANSACTIONS

Serial	Coin	Public key
0	1@A	U1@A -> U2@A
1	2@A	U2@A -> U3@A
2	3@A	U3@A -> U4@A
4	1@B	U1@B· -> U2@B
5	2@B	U2@B-> U3@B
6	3@B	U3@B -> U4@B

Figure 4.9: Transactions of the #N block of the over ledger shown in 4.8

4.2.7 Blockchain of Blockchains

If there are more blocks of the same blockchain we cannot model all the transactions as they happen at the same moment because this may lead us to put in the same set conflicting transactions. In fact, there exists a strict order relation between transactions in different blocks of the same blockchain and in this way we are considering them all equal.



Coin	Public key
1@A	U1@A -> U2@A
1@B	U1@B -> U2@B
1@B	U2@B -> U3@B

Figure 4.10: Conflicting transactions in the same over block

In the 4.10 there is a Hasse Diagram of the order of the blocks into that shows a cross ledger transaction between the blocks $\#2A$ and $\#3B$. In this case, the transaction in the block $\#1B$ transfers the coin 1@B from U1@B to U2@B; the transaction in the block $\#2B$ moves the same coin from U2@B to U3@B. Now there is a clear, strict order relation between these two transactions. We cannot ignore this order and consider these transactions happen at the same moment because the transfer that moves the coin from U2@B is not even valid since U2@B does not own the coin yet.

Hence, the proper model of the over ledger should treat the blocks or transactions set as a poset rather than a regular set. It leads the block in our model to contain a little piece of chains. We obtain what in computer science is known as a list of lists, in this case, a blockchain of blockchains.

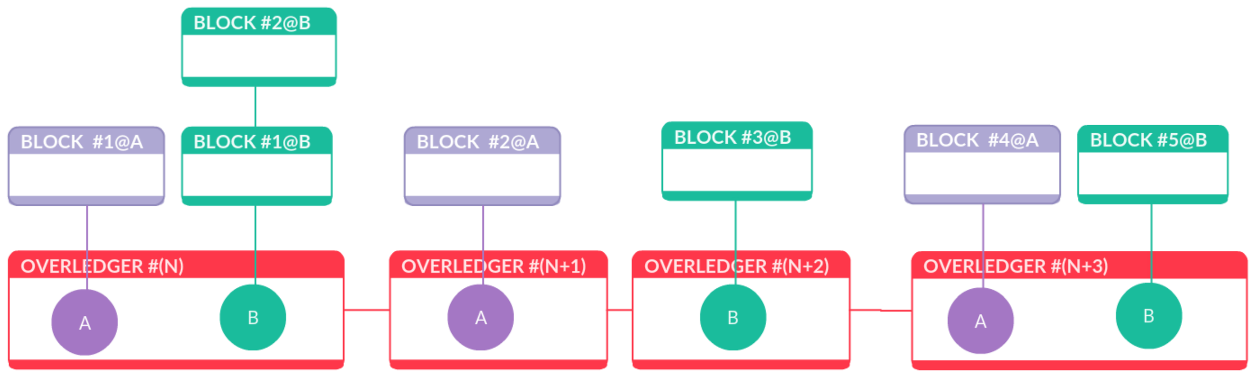


Figure 4.11: Blockchain of blockchains representation of the overledger whose Hasse Diagram is in 4.10

4.2.8 Fork Issue on multi-ledger application

We described blockchains as a list of valid block and we said that there are accepted rules to accept whether a new block is valid. For example in PoW you are entitled to add a new block if you solve a puzzle that changes for each new block. However, it can happen that more users solved the puzzle almost at the same time and there is no way to decide which one was the first or a group of users change the ledger validation rules accepting different blocks and creating a new, different branch. Upgrade a blockchain leads often to a fork between the branch of users that have upgraded the protocol and ones that have not. Whatever is the reason, forks can compromise out business logic. Consider a branch that contains a set of messages and the second one that has a completely different set or with a different order. This problem is solved in single ledger environment with some techniques that are not applicable in our context. Please consider that in a fork situation even if the branches are in conflicts with each other, they are consistent alone. Therefore way to solve this problem is to choose branch among the others as the valid one.e one

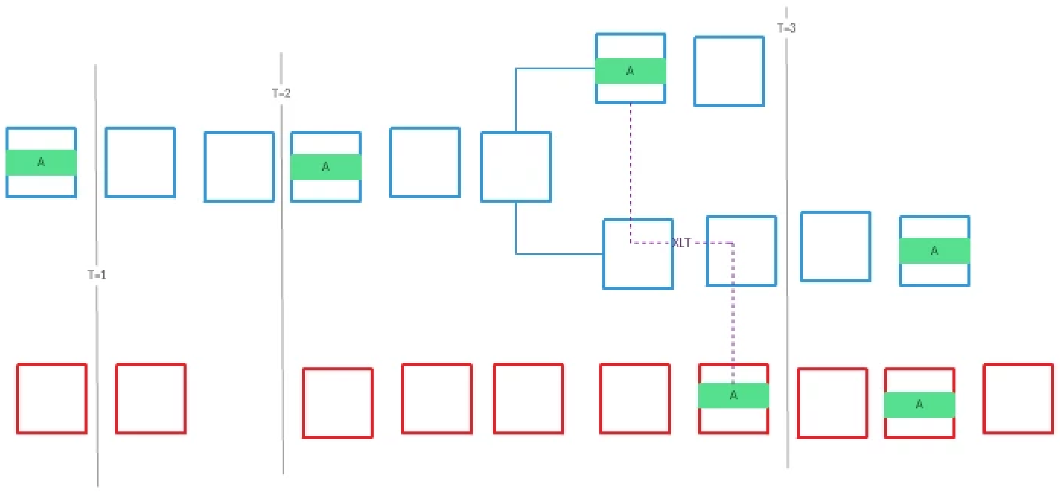


Figure 4.12: A fork happened in one of the ledger of the overledger leading to inconsistency

For example, some of the PoW ledgers consider the branch with more block the valid one. Let's suppose (figure 4.12) that our overledger application append some transactions in one of this branches because it is not yet aware of the forks. If the surviving branch has not the transaction we appended on the other branch or it has been appended with a different order, the ordering and therefore the business logic is compromised. It is because, even if the single blockchain is fork-resistant and can backup in a consistent state, we cannot be sure that the restored consistent state is the one which is consistent with our logic and with the other ledger involved in the same overledger.

Verification Blocks to preserve the order

We introduced a cross ledger consistent to solve this issue. Given a transaction in the verification block n this comes after > every transaction in a transaction block m (in the diagram 4.13 m is represented by n-1) with m < n. To order the transactions within an application, the application scans the ledgers involved and places transaction hashes which are compliant to the Applications Blockchain Programming Interface (BPI) into Virtual block (Verification Block) in the application,a hash pointer to the Verification block is then written to the blockchains that form part of the application. After a given interval , which is dependant on variables such as number of application transactions, block height etc. The application re scans the blockchains involved back to the verification block any new BPI compliant transactions which have been appended to the blockchains that are part of the application are then appended to the Verification block which is updated on the blockchains involved.Note there is no order among transactions in the same block or among invalid transactions. This as the verification block is keeping the order across all chains involved in the application of hashes which have been appended after the consensus has been reached. On rare occasions the application may write a transaction to one of the blockchains involved and this transaction may be invalidated by the consensus mechanism preforming a fork. The above ordering solution solves this by appending the last known verification block which is common across the blockchains involved to the forked chain. The application then scans back across the blockchain re-appending any transactions which have been invalidated by the fork. We can model the verification blocks as the overledger ones.

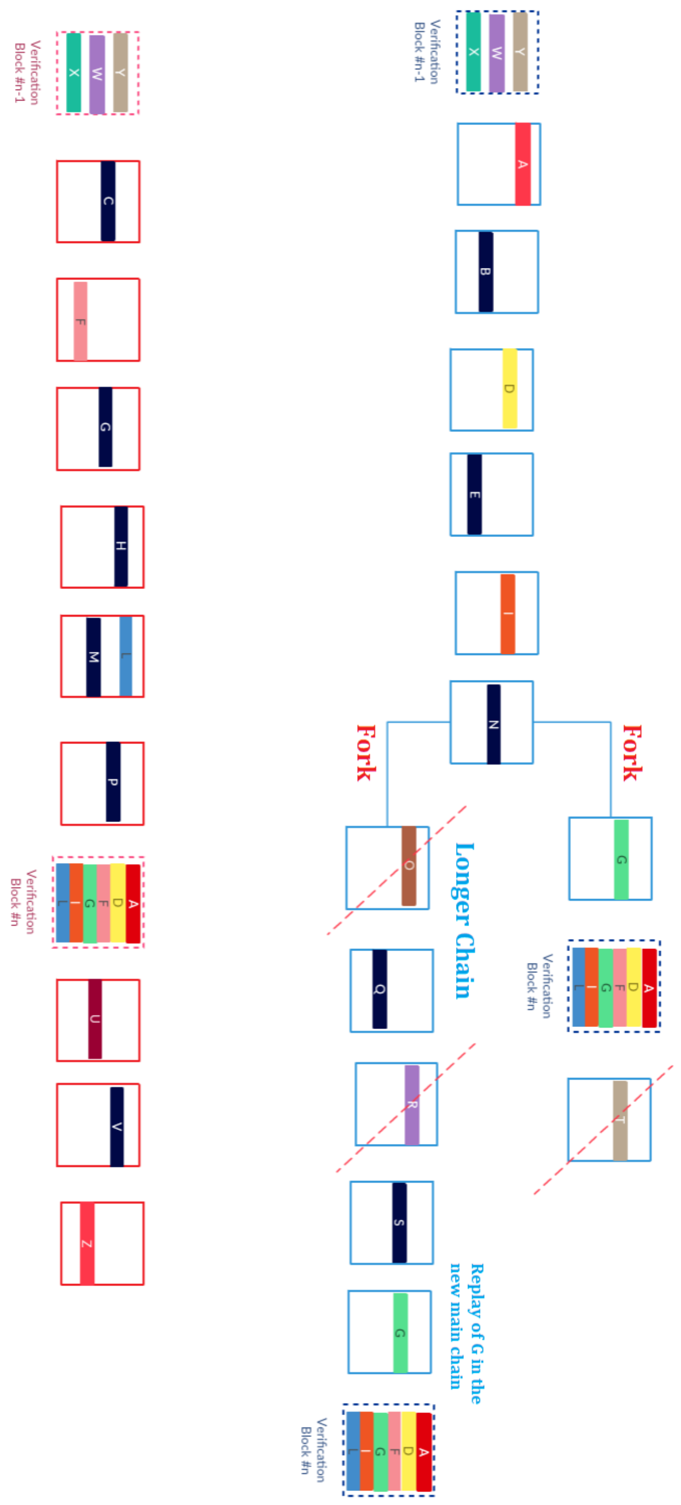


Figure 4.13: Ordering layer through a cross chain consensus mechanism

Chapter 5

Cross Ledger Transactions

'No man is an Iland, intire of it selfe; every man is a peece of the Continent, a part of the maine...'

John Donne, *Devotions Upon Emergent Occasions, Meditation 17*

In a distributed environment it is extremely challenging to build applications. On the one hand, we want our applications to guarantee validity in the event of failures and errors. On the other hand, we want high availability and openness. It becomes harder in an environment where nodes behave according to the BAR model (Byzantine, Altruistic Rational). In this context you would like to develop a system that is ACID compliant Byzantine Fault tolerant. Achieving ACID and BFT in ledger logic is hard because you need to consider failures, errors and malicious behaviours in all the sub chains and their interactions. In the section 4.2.6 we introduce Inter Ledger Transactions and study their importance to build an order between blocks in different blockchains, therefore, an order between transactions.

5.1 Blockchain and transaction properties

In computer science, the acronym ACID stands for Atomicity, Consistency, Isolation and Durability. These properties guarantee that transactions always work in the way that is compliant and valid with the business logic. It is not easy to satisfy all these properties and systems that require high availability of data and high scalability, rather use the BASE property of eventual consistency (Basically Available, Soft State, Eventually consistent). BASE systems do not guarantee consistency in charge of availability. It is necessary because of the Brewer's CAP theorem. CAP theorem states that a distributed system cannot provide all these three properties together:

- Consistency: every time a user manages to read data he will receive the most recent value of that data, but it is not guaranteed to receive an answer.
- Availability: every time an user will try to read data he will always receive a reply but is not ensure that he receives the most recent values.
- Partition tolerance: the system will work even after many messages were lost among two different partition of the system.

BASE architecture is used in domains where there is a huge demand for data, and approximate answers can be tolerated as well as to use stale data. E-commerce sites can show the last item to many users, and it can happen that many users buy the same item. It is not a problem for the E-commerce to sell the item to one of them and to apologise with the other ones. In contexts where users are dealing with asset transfers, there is a need to assure validity and all the ACID properties. Blockchains do not guarantee in a deterministic way all the ACID properties, and it can be rather considered.

5.2 ACID and Blockchain

Atomicity

Each transaction is not breakable into smaller operations, and its execution is either or succeeds or aborts. Blockchain always guarantees atomicity since a transaction can be only present or absent in a block. Therefore, there are no intermediate states where the operation can fail. It is true also for the smart contracts because even if it is true that there is no a precise state of that assets in the contract, the constraints that rule those assets are either or committed or aborted.

Consistency

Before an operation, the system is in a state compliant with the business logic, and after the operation, the system is in another state compliant with the business logic. It means to assure data integrity among data within the system. Please notice that consistency here has a different meaning from the consistency in the Cap theorem 5.1. Blockchain guarantees consistency, in fact, every time transactions are validated a block is appended. Appending a new block should not lead the blockchain to an invalid state because the consensus mechanism assures that it is a valid block that brings directly the blockchain to a valid newer state.

Isolation

Each transaction must be executed in a way that will not interfere with others having the same result of a system that executes all the transactions sequentially. Block's strict order in blockchain guarantees the isolation property.

Durability

Commit a transaction produces changes that will be persistent. Blockchains do not guarantee this property in a deterministic way because there is the case that many miners claim to have found the next block.Until one of the two forks becomes sufficiently long to be unlike to be replaced from a longer chain, transactions within blocks are not persistent.

5.3 BASE and Blockchain

Eventual consistency

According to this model, different nodes can store different and conflicting states of the logic. In a sufficiently long period, nodes detect conflicts and solve them. This means that the nodes will eventually give the most updated version of data. This architecture enforces liveness because nodes do not need to be sure that their data version is the most updated and they can directly answer the request with the best answer they can give.

SEC: Strong eventual consistency

In some context, data updates bring the state from one point A to the same point B regardless the order of transactions.In other words, if two nodes receive the same messages in the different order they are in the same state. A counter that receives the same increment operations and decrement operations will be in the same state (value) regardless the order of these operations. This property is not true for transactions and can lead to an invalid state because the set of transactions is a poset and we cannot ignore the order as shown in 4.10. However, blocks in the blockchain have the order that guarantees consistency among transactions embedded in their format. If a node receives the same blocks, but in a different order he can rebuild the proper order using hash pointers. This means that we can consider the Blockchain as a Strong eventual consistency system that can be considered ACID with a probability that can be estimated proportionally to the number of blocks that succeed the block of interest.

5.4 Cross Ledger Transaction consistency

Even if blockchains do not support the ACID transaction, business logic transactions need to work in an ACID environment. In 5.3 we saw how to approximate blockchain transactions to ACID transactions with a certain probability due to the number of the confirmations. Achieving ACID in cross ledger transactions is even harder since we cannot only rely on block confirmations. Please consider that not only blockchains need the confirmation time (e.g. Ripple). However, we need to consider the worst scenario model.

Cross Ledger Transaction and ACID

Atomicty

In XLT (Cross ledger transactions) we need to record the data in all the blockchains interested in the exchange. In the easiest case like in 4.6 the transaction need to be recorded in both the domains of blockchain A and B. The transfer operation is not more atomic because it consists of two sub-operations.

Consistency

It is possible that an XLT is recorded in only one of the BC breaking the consistency among the different blockchains. In fact if the Blockchain a records the transfer of an asset in the domain of the blockchain B it will consider the asset in B domain conversely the blockchain B will consider that asset still in A. The two blockchains do not agree and since an asset can exist in only one domain there is a data integrity violation.

Isolation

XLTs do not guarantee isolation. Let's suppose on the blockchain B there is a transaction that moves an asset from A to the user U1@B. Suppose that this transfer failed on the Blockchain A. The user U1@B can use his partial information to move an asset he does not own. This is an example of what can happen if acting upon uncommitted information.

Durability

Since XLTs are based on blockchains, the durability property is not guaranteed as well. We can estimate a probability that is the product of the confirmation probabilities of all blockchains.

5.5 Two phase commit and XLT

Two-phase commit also known as 2PC is a distributed algorithm that validates transactions in distributed environments. All nodes interested need to express a commit or an abort to a coordinator according to a time scheme. If the coordinator receives all commits from the nodes will validate the transactions otherwise will abort it. It is called two-phase commit because a commit can be reached after two phases. In the first one (voting phase) all nodes express their intention to commit or abort. In the second phase (commit phase) the coordinator will commit the transaction only if there are no abort or error messages. In this scenario also a timeout is considered an error. The problem with XLTs is that we need to distribute the coordinator's job.

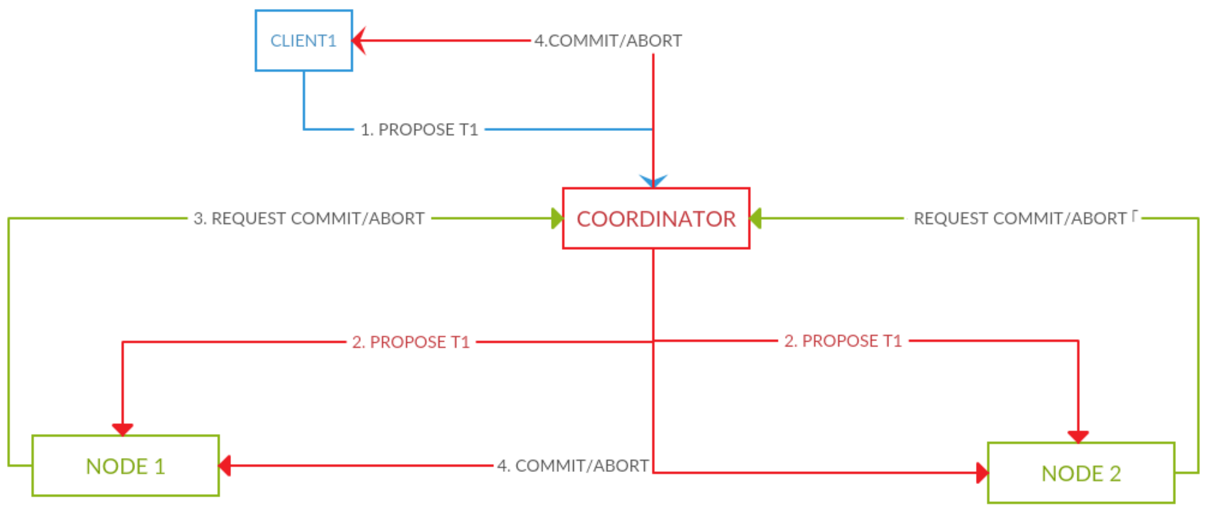


Figure 5.1: Two phase commit scheme

In the proposed scheme the role of the coordinator is made by the two entities that exchange the asset. They need to post a 'ready' message that acts like a commit request on all the blockchains interested. When they succeed to add a 'ready' message on all the chains, they then confirm and commit the transaction on the chain where the transfer started. All the messages that are 'ready 'have a hash pointer to the proposed message to prove the user's intentions. The final commit message contains all the hash pointers of the' ready' messages. This scheme locks the coin until a successful transfer. We will introduce later more complicated models that include the possibility to abort only in the chain where the transfer started or in all of the blockchains, and we will show pros and cons of these solutions.

Figure 5.2: Two phase commit scheme on two ledgers

5.6 XLT FORK and Verification Block

The two phase commit schema we have presented can guarantee the atomicity for transactions that require more than one operations at the low-level (transaction one). However it is not fork-consistent. It is good because it can be useful when you want to reach consensus from users that are not in the same ledger and that have not visibility on the others.

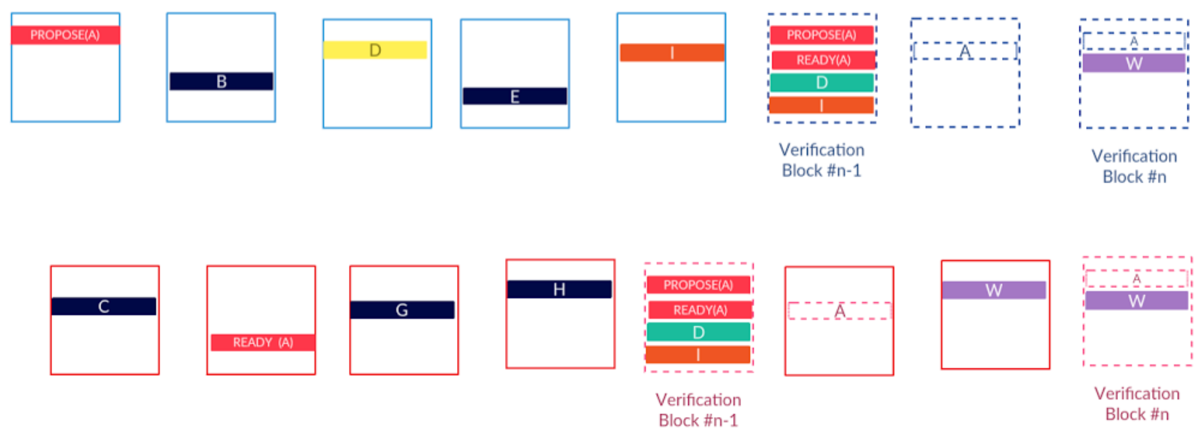


Figure 5.3: Fork during in a XLT schema. Dashed blocks are the ones appended by the application. Dashed transactions are the one which proposed by the application

The figure 4.12 in the previous section, shows two blockchains (red and blue). In the blue blockchain a fork happened. In the upper branch there is a XLT transaction between the blue chain and red one. The lower branch which does not contain the XLT transaction becomes the main branch (longer one) leading the multi-ledger application to a inconsistent state. We can solve this issue in the same way we did with fork for the ordering in 4.2.8. In the figure **??** the atomic transaction is the red one. A user in the blue blockchain we will call him Blue, proposes a transaction/message to a user of the red blockchain (Red). Red knows about the intention of the user of Blue because of the proposal message and propose a Ready message. Notice that Blue can notify Red by sending a message out of the chain or as in this example, the application replays the Propose message in the red blockchain.

In terms of transaction address :

- Blue chain Propose A : Blue@bluechain to App@bluechain
- Red chain [optional] Propose A : App@redchain -¿ Red@redchain
- Red chain Ready A : Red@redchain -¿ App@redchain
- Commit A : App@bluechain -¿ App@bluechain
- Blue chain Verification block : App@bluechain -¿ App@bluechain
- Red chain Verification block : App@redchain -¿ App@redchain

Chapter 6

Application Layer

'An abstraction is one thing that represents several real things equally well'

Edsger W. Dijkstra*, quoted by David Lorge Parnas: Use the Simplest Model, But Not Too Simple*

In this section, we will explore the application layer the upper part of our reference architecture. In this domain, there are several isolated applications that have their business logic that is independent of the lower component. Applications have the chance to communicate each other putting messages on the message layer. If these messages are compliant with the filtering rules of the other application, they can flow through the filtering layer to the application layer. Applications can implement communication mechanisms that do not use this scheme. This scheme allows anonymous user to send a message to the application if it is compliant with its business logic.

6.1 Overldger applications

In the previous sections we explored from a theoretic point of view how to build an overledger by adding messages as meta information on different ledgers, giving them an order and make them part of a more complex application business logic. In this section we will describe what are the steps to perform a basic transaction, i.e. append a message in a particular Over ledger.

6.1.1 How many overledgers?

If we want to give a formal definition to the overldger, it is a sorted list of messages which satisfy a set of unambiguous properties. A set of properties define which is the valid format of a message, how to build its fingerprints and what requirements a transaction should meet to host the fingerprint of a valid message. These list of these messages defines what we called the Message-Filtering layer of the overledger. Another set of rules determines how to sort the valid messages in a sorted list we called Sorting layer of overledger. A system that uses an overdleger reacts to these sequence and can change its state. Any change in one of these two sets of rules results in a different list of messages or a different permutation. More systems, with different control logic, can share the same overledger if they have respect the same rules.

6.1.2 BPI - Blockchain platform interface

Systems (e.g. applications) to use an overledger need to define the two set of the rules. These rules determines a wire protocol to interact with the system and with the other users. The rules can be as follows:

- Accept messages that can be validated by specific schema
- Accept messages only certain pattern sequences (e.g. in a two phase commit we want the 'propose' before the 'ready' message)
- Accept messages only if their fingerprint (hash) has been appended on a particular set of ledgers.
- Accept messages only if their fingerprint has specific source and recipe addresses.
- Accept messahes only if their fingerprint is spending at least a certain amount of criptocurrency.

Notice, that an application can define more complex messages. It can define combination of them (e.g. different messages can have different validation schema, different rules on the transactions hosting their fingerprint. The rules can also involve other information of the information for example the script or the contract that they contain. This schema is similar to the one used by systems which exposes APIs. In fact, those systems described at the application level what are the methods to interact with them.They publish a file and a library to interact with the system as it is ADT(abstract data type). Likewise our set of rules describes how to interact with the overledger. Even if the approach is the same, please notice that API are on the application level, our rules are on a message, technology independent level. We call them BPI (Blockchain platform interface). A BPI is essentially a marked up text which represents which define a specific overledger and the rules to add messages to it. The BPI should be automatically read by clients that implements the required methods according to their technology. It would be helpful a Overdleger SKD allowing application to define only the rules (BPI) without taking care at the low level details of the transactions and messages.

6.1.3 The Transaction Journey

This section describes the steps to append a valid message. In this example there is an user (client) that wants to append a specific message on the overledger. Notice that these roles in this example are interchangeable.

- The client read the requirements for the message it wants to appends
- The client build a valid message M1 and calculate its fingerprint H1
- The client build a valid transaction T1(inserting the fingerprint) and proposes it to the systems in charge to propose transactions.
- T1 is appended on an ledger accepted by the BPI
- The client sends the complete message to the application and the transaction T1 that comtains its fingerprint

6.2 Application level responsibility

In the section 3 and 4 we discuss the different roles of the message layer and the filtering layer. We explained that those are logical layers, but we do not explain who is in charge to implement

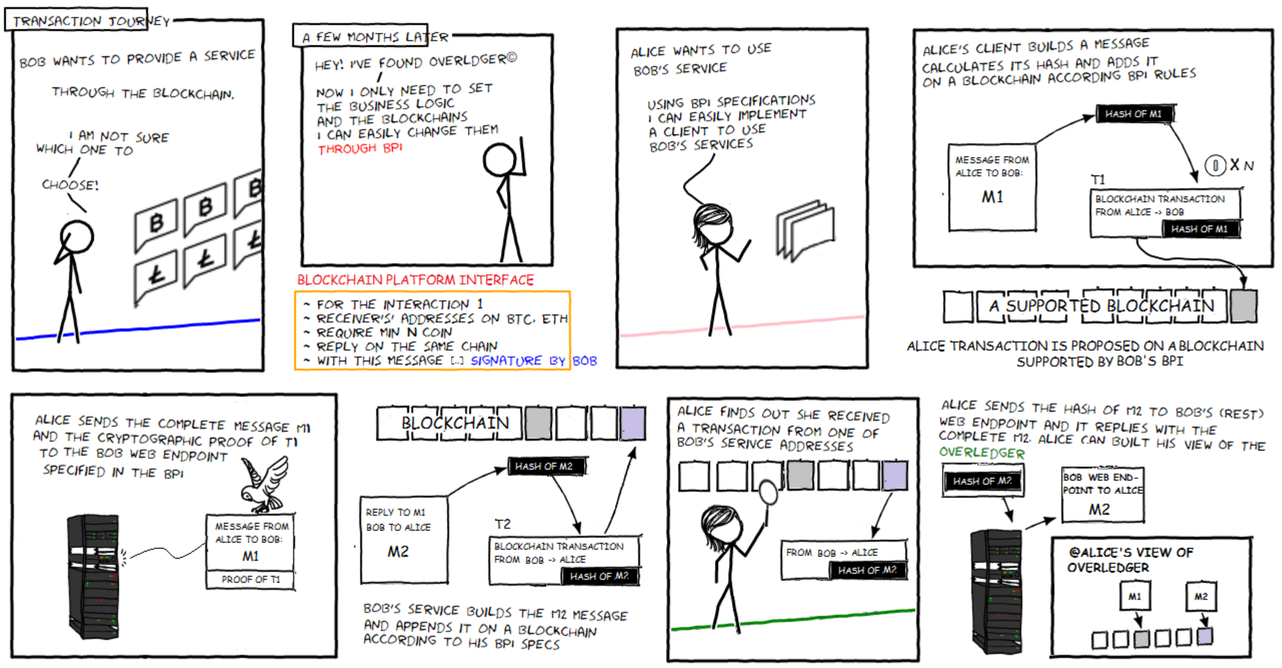

TRANSACTION JOURNEY
BOB WANTS TO PROVIDE A SERVICE THROUGH THE BLOCKCHAIN.
I AM NOT SURE WHICH ONE TO CHOOSE!
A FEW MONTHS LATER
HEY! I'VE FOUND OVERLDGER©
NOW I ONLY NEED TO SET THE BUSINESS LOGIC AND THE BLOCKCHAINS I CAN EASILY CHANGE THEM THROUGH BPI
BLOCKCHAIN PLATFORM INTERFACE
~ FOR THE INTERACTION 1
~ RECEIVER'S' ADDRESSES ON BTC, ETH
~ REQUIRE MIN N COIN
~ REPLY ON THE SAME CHAIN
~ WITH THIS MESSAGE [..] SIGNATURE BY BOB
ALICE WANTS TO USE BOB'S SERVICE
USING BPI SPECIFICATIONS I CAN EASILY IMPLEMENT A CLIENT TO USE BOB'S SERVICES
ALICE'S CLIENT BUILDS A MESSAGE CALCULATES ITS HASH AND ADDS IT ON A BLOCKCHAIN ACCORDING BPI RULES
MESSAGE FROM ALICE TO BOB: M1
HASH OF M1
0 X N
T1
BLOCKCHAIN TRANSACTION FROM ALICE -> BOB
HASH OF M1
A SUPPORTED BLOCKCHAIN
ALICE TRANSACTION IS PROPOSED ON A BLOCKCHAIN SUPPORTED BY BOB'S BPI
ALICE SENDS THE COMPLETE MESSAGE M1 AND THE CRYPTOGRAPHIC PROOF OF T1 TO THE BOB WEB ENDPOINT SPECIFIED IN THE BPI
MESSAGE FROM ALICE TO BOB: M1
PROOF OF T1
BLOCKCHAIN
HASH OF M2
REPLY TO M1 BOB TO ALICE M2
T2
BLOCKCHAIN TRANSACTION FROM BOB -> ALICE
HASH OF M2
BOB'S SERVICE BUILDS THE M2 MESSAGE AND APPENDS IT ON A BLOCKCHAIN ACCORDING TO HIS BPI SPECS
ALICE FINDS OUT SHE RECEIVED A TRANSACTION FROM ONE OF BOB'S SERIVCE ADDRESSES
FROM BOB -> ALICE
HASH OF M2
ALICE SENDS THE HASH OF M2 TO BOB'S (REST) WEB ENDPOINT AND IT REPLIES WITH THE COMPLETE M2. ALICE CAN BUILT HIS VIEW OF THE OVERLEDGER
HASH OF M2
BOB WEB END-POINT TO ALICE M2
@ALICE'S VIEW OF OVERLEDGER
M1
M2

Figure 6.1: The transaction journey in Overldger: representation of a simple transaction

these features. Filtering messages requires a deep knowledge of all the blockchains included in the application, for this reason, it would be convenient to build open source libraries that help the application developer to use a high level function to interact with the blockchain.

```
import bitcoin as btc
import ethereum as eth
import json
import hashlib
import requests

# Message definition#
jstring = {'app_id'=1, message'='Hello world', 'client'=1}
msg = json.dumps(jstring, sort_keys=True)

hash_function = hashlib.sha256()
hash_function.update(msg)
hash_message = hash_function.digest()
```

```
r = requests.post(server_address, json=msg)

eth.add_msg_out_of_the_chain(hash_message)
btc.add_msg_out_of_the_chain(hash_message)
btc.add_msg_out_of_the_chain(hash_message, mode='OP_RETURN')
```

Notice that one of the parameters of the json'.dump function requires the keys of the JSON to be sorted. This is mandatory to be sure that same json with a different order key produces the same hash.

In this example, the code takes all the responsibilities for all the layers. This is an example of a write operation of a client that sends a message to a particular application ($app\_id = 1$). In this case, the lines are both related to the filtering layer. The definition of a particular message format, the values of the particular message instance, the chosen hash function and the mode how the message is added to the ledger. The last two lines of the code show that the Bitcoin chain can allow a different way to add information to its ledger. In the last line, the optional parameter mode specifies which one must be used rather than the default option. The line that performs the post-operation send through the network the entire message. The server is responsible for checking if the message is valid and in this case if the hash of that message is on the blockchains of interest with the proper transaction parameter (addresses and coins). To check if the message is in the chain, the application needs to perform a scan of the newer blocks and react whether it finds or not the message. This can lead to solutions that need a timeout and complex protocol. If the server finds the hash message, it can react because in the JSON file there is all the information of the receiver. In this case, there is the field 'client' in the message equals to one. That number can refer to a client address or to a set of configuration to allow the server to reply to it with this schema, adding messages on blockchains. Another design solution can assign the responsibility to check the hash to the client that can scan the interested ledgers until it finds the transaction and can prove giving the transaction's hash pointer (e.g. Merkle tree proof) to the server. In this case, the server can quickly check the lowest details and does not need to pull all the interested chains.

```
import quant.app_builder as q
```

```
app = new q.client_application()
#in conf.txt there are all the information
#about the Filtering Layer
app.set_configuration(./conf.txt)
jstring = {'app_id'=1, message'='Hello world'}
msg = json.dumps(jstring)
app.sendmessage(msg)
```

In this last example, we introduced in the import statement a library 'quant' that takes the responsibility of the filtering layer' tasks. The application loads the configuration file that can be hosted on the local storage or can be downloaded every time to guarantee consistency between the application upgrade and the client application. In this case, the developer needs only to focus on the business logic of the client.

6.3 Writing Operation in the Overledger architecture

In the last section, we gave two example on how the application could look. The two listed examples two client applications that send a 'hello world' message to a server application. The write application in the application layer consists of building the messages and publishing on the proper chains according to the server configuration. Let's suppose these are the specification of the server application

- Use Bitcoin OP_RETURN field.
- Use Ethereum Bytecode field.
- Publish the information on both blockchains (parallelization).
- Use ADDR1 as destination for Bitcoin transaction
- Use ADDR2 as destination for Ethereum transaction
- Accept transaction more than x for Bitcoin
- Accept transaction more than y for Ethereum

- Message format JSON
- Validate message JSON with Schema1

If we want to use the approach of the-the second piece of code 6.2 we need to build a configuration schema that can be later loaded from the library and turned in code like 6.2.

```
{
  "application name": "Server Example",
  "namespace" : "UCL_CBT",
  "Owner": "Gaetano Mondelli",

  "Messages supported" : [
  {
  "name of the message": "Send message"
  "blockchain message" : true
  "out of the chain msg" : true,
  "hash function" : "MD5"
  "schema" : "schema1"
  "transaction mode":"redundant",
  "Blockchain supported": [{
    "name": "Bitcoin",
    "mode": "OP_RETURN",
    "source": null,
    "destination": "ADDR1",
    "coin_amount": "x",
    "need the proof": false,
    "need the source": true
  }, {
    "name": "Ethereum",
    "mode": "Bytecode",
    "source": null,
    "destination": "ADDR2",
    "coin_amount": "y",
    "need the proof": false,
  "need the source": true
  }]
  },
  {
  "name of the message": "Send proof"
  "blockchain message" : false
  "endpoint" : SERVER1_ADDRESS
  "schema" : "schema2"
  }
}
```

The library will load this information and when the function app.sendmessage(msg) is called the application will execute the following lines of code.

```
hash_function = hashlib.md5()
hash_function.update(msg)
hash_message = hash_function.digest()
endpoint = SERVER1_ADDRESS
eth.add_msg_out_of_the_chain(hash_message,dest="ADDR2",
coin="y+1")
btc.add_msg_out_of_the_chain(hash_message,
mode='OP_RETURN',dest="ADDR1",coin="x+1")
# This is polling for the proof of the transaction
r = requests.post(endpoint, json=msg)
```

6.3.1 Dependency injection and Overledger

6.4 Reading Operation in the Overledger architecture

In some context, it is required that the application (server or client) perform a read of the blockchains to react properly. The reading part of an over ledger application is in charge to read the blockchains, building the messages, filtering and ordering them and finally pass them to the application level.

Let's suppose that in the previous example the application "Server Example" has one of the blockchain's parameter "need the proof" equals to true. In this case, the client needs to wait for their transaction to become valid ones in the chain.

```
hash_function = hashlib.md5()
hash_function.update(msg)
hash_message = hash_function.digest()
endpoint = SERVER1_ADDRESS
eth.set_msg_out_of_the_chain(hash_message, dest="ADDR2",
coin="y+1")
btc.set_msg_out_of_the_chain(hash_message,
mode='OP_RETURN', dest="ADDR1", coin="x+1")
# This is polling for the proof of the transaction
while (p1 = eth.publish_message() and p2 = btc.publish_message())
proof_message = {"proof btc":p2, "proof eth" : p1}
msg = addprooftomessage(msg, proof_message)
r = requests.post(endpoint, json=msg)
```

In this case, the client application cannot defer the publish of the message and need to wait for the response with the hash to prove where there is the transaction with the wanted hash. In this case, the reading part consists only in polling the new block of blockchain to find the proof of the transaction. Some application can require scanning piece of blockchain on the first boot to find the piece of information in the blockchain that was published before the launch of the application. Some application can search for more complicated information, looking field that handles the blockchain scripts. This information can allow a smart contract to be valid or reinterpreted to the application level. For this reason, we need a library that includes all these situations, and that can boost the interest of developers in this architecture. In this last example, we propose a functional approach to the reading problem in a specific case. The application scanned a part of the Bitcoin blockchain to find one information and then is waiting for a hash in OP_RETURN field of Bitcoin.

```
import bitcoin_library as btc
```

```
err =false
l = list(filter(lambda x: x.destination == 'ADDR1',
btc.gettransactionblocks('OP_RETURN',45000,5001)))

t1 = time.now()
while( count (l = list(filter(lambda x: x.destination == 'ADDR1',
btc.gettransactionblocks(5001,null)))) == 0):
    #sleep(tx)
    t2 = time.now()
    if t2-t1 > threshold :
        err = true
        break
```

In this case, the application is filtering all the transaction in the Bitcoin chain between the heights 45000 and 50001 looking for transactions that have as destination ADDR1. It is essential that the function that gives the list of the transaction has a good cache system to do not perform reading operation on the chain for each request. The second part is not checking for that information in a specific piece of the chain but since the second parameter is null is waiting until there is at least one transaction record with the destination equals to ADDR1'

6.5 Case studies

6.5.1 CQRS-ES and Overledger

Command Query Responsability Segregation

CQRS (Command Query Responsability Segregation) is an architectural pattern introduced by Greg Young. This pattern uses different models for writing and reading the model. The writing model should manage the command operations that following the CQS (Command Query Separation) vocabulary can update the model and must not return a value. The reading operations should handle the query operations that must not change the system state (no side-effects). In some domain this pattern can be useful for breaking down the complexity of a system especially where reading operations and writing operations have different scale.

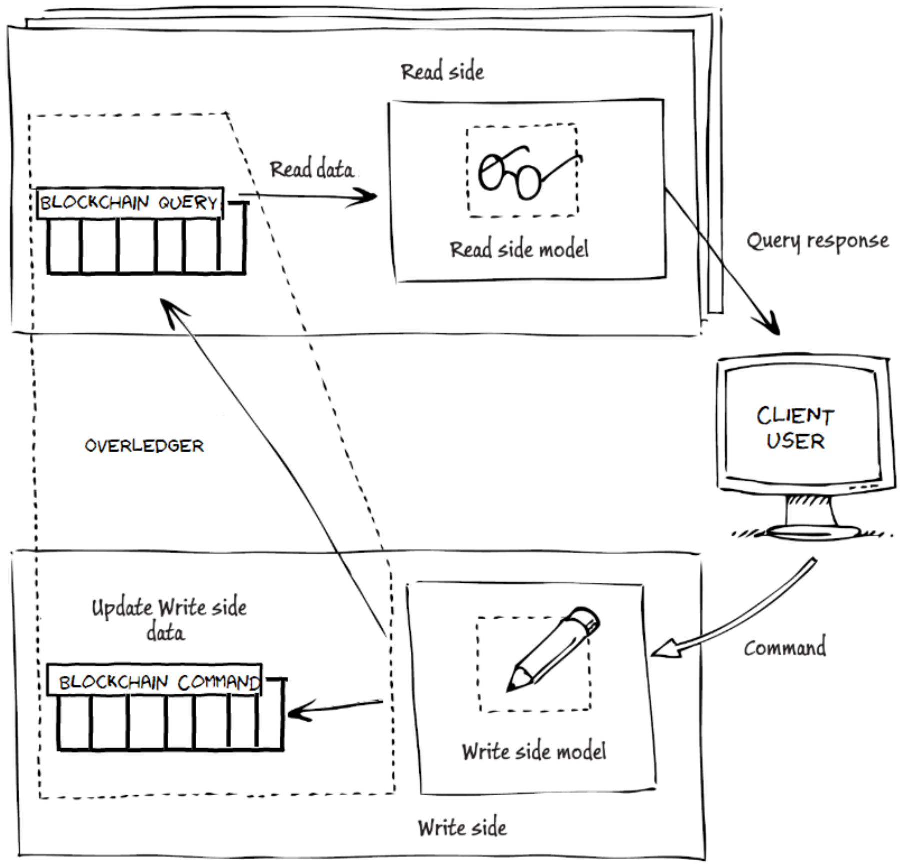


Figure 6.2: CQRS Architectural Schema for Overledger

Overledger can be a good fit for this pattern implementation. We can use two or more different blockchains some for appending commands and some other for querying the status. This can be useful not only because the application may have different scale for reading and writing but because different blockchains have different level of security and different throughput. For example we can use faster blockchain to collect the commands and a more secure (and probably slower) one to store the valid transaction in the application logic or use different blockchain to show different type of aggregate according to the clients needs. For example this blockchain could store only client state or verification blocks.

6.5.2 Event Sourcing and Overledger

Event Sourcing is a pattern that builds the status of the system scanning all the events that have changed it. When a new event happens rather than updates the state of the system, events sourcing architecture appends it to the event stream. The state is calculating by replaying all the events. Since appending events is a single operation, Event Sourcing is inherently atomic. The way how the state is calculated has a lot of similarities with the blockchain technology. In the figure 6.4 events are appended to one blockchain and different domains of the applications retrieve those event data for many reasons. For example one domain can be interested in storing data for ETL, one for optimising searches and another one for processing the event and stream the output in another blockchain. The similarities between this pattern and the mode of operation of blockchain makes its integration with overledger natural and recommended. In the literature CQRS and ES are often combined and used together (CQRS-ES), notice that this is still applicable with overledger.

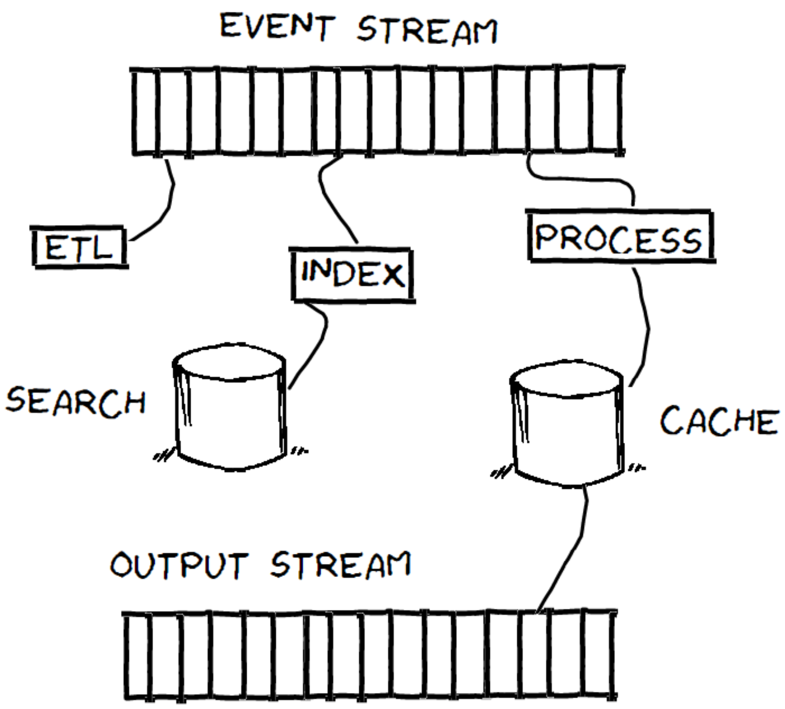


Figure 6.3: CQRS Architectural Schema for Overledger

6.5.3 Encapsulation and InternetOverLedger

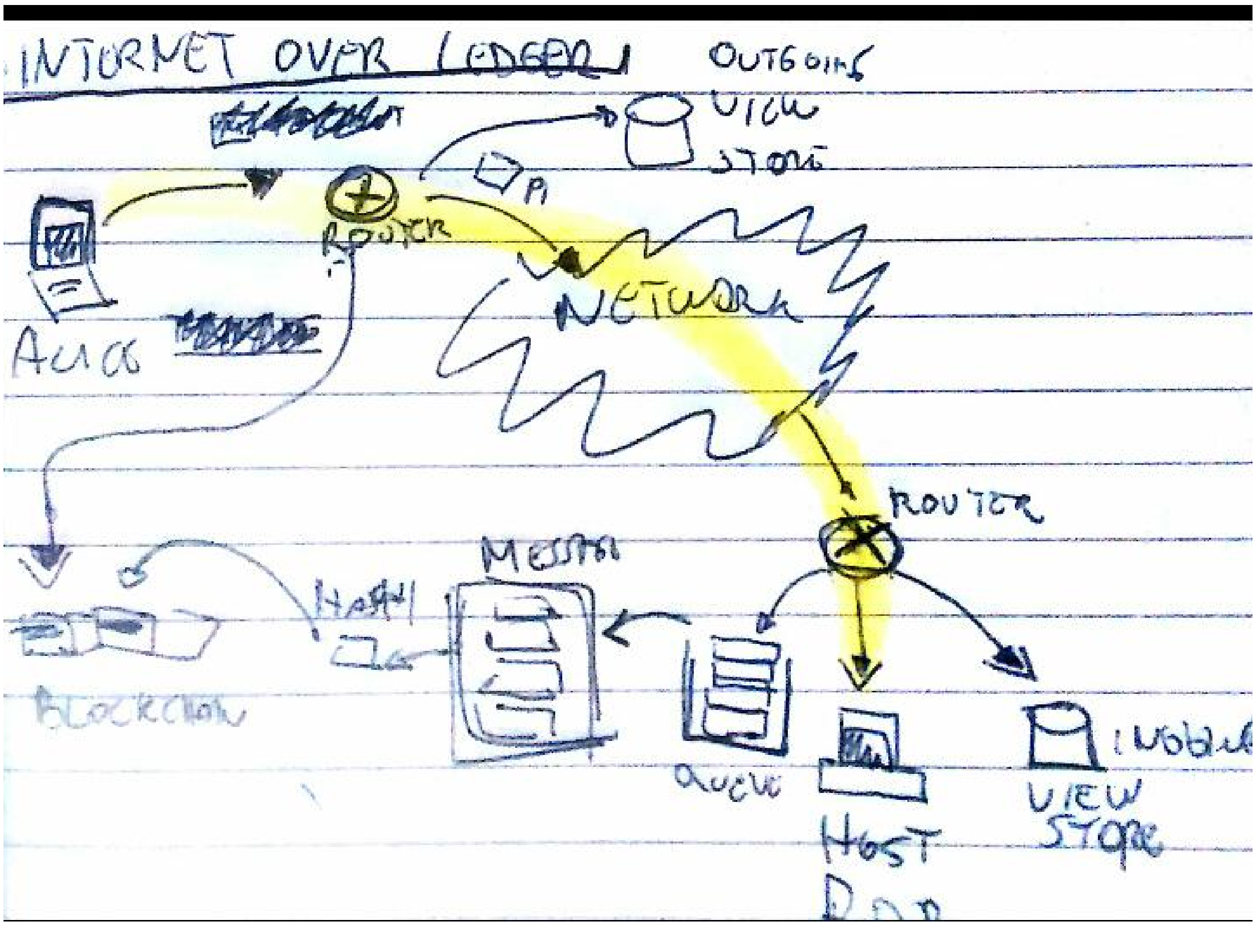


Figure 6.4: Internet Packets encapsulated on out-of-the-chain messages to trace a stream communication

Bibliography

[AN15] Edward Felten Andrew Miller Steven Goldfeder Arvind Narayanan, Joseph Bonneau. Bitcoin and cryptocurrency technologies. 2015.

[MSM17] Pierre Karpman Ange Albertini Marc Stevens, Elie Bursztein and Yarik Markov. The first collision for full sha-1. 2017.

[PT17] Claudio J. Tessone Paolo Tasca, Thayabaran Thanabalasingham. Ontology of blockchain technologies.principles of identification and classification. 2017.