
**Università degli Studi di Pisa**

**Dipartimento di Informatica**

Corso di Laurea Magistrale in Informatica

# Authenticated Cross Chain Data Transfer on EVM-based blockchains

*Candidato*:

Gianmaria di Rito

*Relatori/Relatrici*:

Prof.ssa Laura Emilia Maria Ricci

Dott. Damiano Di Francesco Maesa

Anno Accademico 2021-2022

# Contents

# Chapter 1

# Introduction

Nowadays, more and more attention and funds are being invested in the area of blockchain technology [1]. For example, many blockchain-based applications such as NFTs have recently come into the spot light. These are smart contract that certify the ownership of assets, for instance, a digital artwork. More generally, blockchain technology has been the technological enabler behind the new decentralized version of the web named Web 3.0. In this new paradigm, web applications are replaced by DApps, distributed applications that rely on blockchain protocols to achieve decentralization.

Given the increasing diffusion of novel applications, the existing blockchain systems have become increasingly congested with transactions produced by users and services eager to participate in the community and to exploit its novel capabilities. This trend has caused a heavy strain on deployed systems during the last few years. As the number of transactions increases, blockchain protocols experience a degradation of performance and, consequently, user experience. This is mainly reflected in decreased transactions throughput and increased transaction fees. Scalability is, in fact, a well known problem of blockchain protocols, and much work has been produced to try and amend it.

To solve this problem without radically changing the underlying protocols, researchers have proposed so-called Layer-2 solutions aimed, in various ways, at improving the scalability of the system. Examples of these solutions paradigms are *side-chains*[2], *cross-chains*[3] and *rollups*[4].

This thesis is positioned within this area of solutions aiming to increase scalability, focused on the Ethereum protocol. In particular, it addresses the problem of communication between Ethereum side-chains based on the Ethereum Virtual Machine (EVM) model. The goal of side chain based approaches is to split the computational load of a congested main chain between external side chains pegged to the main one. This allows for the

execution of off-chain transactions (on the side chains) easing the burden on the main chain, while keeping the results accessible to and secured by the main chain.

In this ecosystem of different chains sharing information between each other, this thesis concerns the study of authenticated query protocols to allow trusted cross-chain communication between different Ethereum side chains and the main network. Query authentication is a topic typically related to the interaction between light nodes and full nodes of a blockchain network, to allow resource constraints nodes to fetch information from and participate to a given blockchain protocol. Our work stems form the observation that the authenticated data structures and verification objects used by these protocols can also be exploited to securely transfer information between different chains.

The exchange of information from one chain to another is similar to the access of information from a light node (in the destination chain of the information) towards a full node that holds the entire ledger (the source chain). In the literature there are many solutions to allow the exchange of information in an authenticated way. For example, Merkle Proof structures exploit the properties of structures such as Merkle trees to provide a proof of inclusion of a block in a given blockchain (inclusion proof), proof regarding the integrity of a block content (integrity proof), or proof of the consistency of a block in two different time moments (consistency proof). We, therefore, believe that using these already available technologies, in a different scenario, can add an interesting solution to the landscape of Layer 2 scaling proposals.

More precisely, the thesis proposes CARONTE, a protocol enabling the trusted exchange of information between two sub-chains *C1* and *C2* of the Ethereum blockchain. Specifically, we want to allow users of *C1* to retrieve variable values contained in a contract saved on *C2*, without having to download all the blocks of *C2* to ensure that the information passed is correct. To implement this protocol, we have developed a smart contract acting as *bridge* between the two sub-chains. The *bridge* smart contract executes all the necessary checks to ensure the validity of the data it receives, i.e. the data exchanged between the two chains. In order to be able to perform validity checks, this *bridge* smart contract has to maintain a compact history of *C2*. To sum up, our proposed protocol workflow follows the following steps:

1. a user on the chain *C1* requests information stored on the chain *C2* by contacting the *bridge* contract;

2. the *bridge* contract sends the request to the *C2* chain;

3. a user on chain *C2* retrieves the requested data and sends it to the

*bridge* contract coupled with a *proof of validity*;

4. the *bridge* contract verifies the data received via *C2's compact history* and the *proof of validity* sent with the data;

5. if the data is valid, it sends it as a response to the user on *C1*.

Our protocol, therefore, offers the possibility of scaling Ethereum by enabling its sub-chains to communicate with each other autonomously from the main network, reducing the load of transactions to be executed on the main chain. In addition, by providing a one-way flow of information, in which the requester has restricted visibility into the source chain, our protocol may also be used by private sub-chains, i.e. chains with a well defined set of entities with access rights.

Another feature of the protocol is that it works directly on-chain so that all meta-data related to the transfers performed will be saved in the contract implementing them. Differently, it has been very common to use external libraries such as web3 to retrieve data from blockchain protocols and validate it off-chain. These, however, do not allow the verification and validation of the retrieved information to be tracked. Not to mention that, in order to perform a secure verification, we would still need several pieces of information contained in the blockchain block headers that would need to be retrieved via web3 as well. Our proposal offers authentication guarantees on the values obtained, external services do not.

To prove our proposal feasibility, we provide multiple implementations of CARONTE. The simplest possible implementation of the bridge contract employs a map to save *C2*'s compact history, but this data structure occupies a lot of space. Therefore, we propose to improve the contracts performances in terms of space used, by employing a sliding window algorithm [5], i.e. remembering only a subset of the most recent history. To further improve space efficiency it is possible to replace the map altogether with a fixed-size array. All three implementations were tested, and our experimental results show how the last solution achieves a good space optimization compared to the two alternatives.

The thesis is organized as follows. Chapter 2 discusses the background needed to understand the work carried out in the thesis. This chapter first discusses the basic data structures used by Ethereum and how they are encoded. It then introduces the structure of blockchains in general and Ethereum in particular, thus dealing with the structure of blocks and the data stored within them. Chapter 3 presents works related to the protocol proposed in this thesis. It explains what Layer 2 solutions are, and which are its most promising proposals. Chapter 4 presents CARONTE, the protocol

proposed in this thesis. The actual implementation of the protocol is then covered in Chapter 5 while chapter 6 shows the tests carried out to evaluate the proper functioning of the *Bridge* contract, the core contract of our implementation. The thesis ends with Chapter 7, which presents our final remarks on this work and a some possible directions to further improve the CARONTE protocol both theoretically and from an implementation point of view.

# Chapter 2

# Background

Since this thesis work is mainly related to the Ethereum blockchain, in this background chapter we will first introduce the main data structures related to its implementation. Then we will discuss what blockchains are and how they are structured in general, focusing specifically on Ethereum.

## 2.1 Trees and Tries

Blockchains use various data structures and algorithms to search for information quickly and securely. The main data structure used by Ethereum and which made it famous is the so-called *Merkle Patricia Trie*[6]. Since it is an merger of the features of multiple data structures already known in computing, let us first explain these structures.

### 2.1.1 Search Trees

One of the most efficient data structures in terms of key search time (*O(h)* in the average case, where $h = \log_2 n$ and $\log_2 n$ is the height of the tree) is the *search tree*. One of its implementations called *Binary Search Tree* was described in section 6.2.2 of The Art of Computer Programming, Volume 3 by Knuth [7]. This tree saves in each node values, such as numerical. Each non-leaf node can have at most 2 branches and this tree must respect these 3 properties:

1. The left sub-tree of a node $x$ contains only nodes with values less than $x$.

2. The right sub-tree of a node $x$ contains only nodes with values greater than $x$.

3. The right and the left sub-trees must themselves be Binary Search Trees.

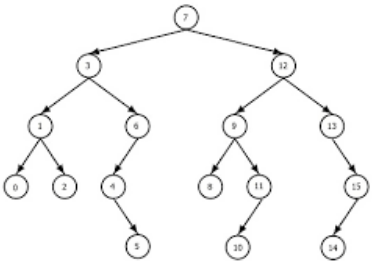

Figure 2.1: Binary Search Tree

A Binary Search Tree can also be seen as a *Prefix Tree* where keys with the same prefixes share the same path from the root to the leafs. A key is searched in such a tree so that the key is iterated from the first to the last character and for every character it is checked if there is a child node in the tree that match the character. The tree traversal descents to the child node, next character is picked-up and the same step repeats.

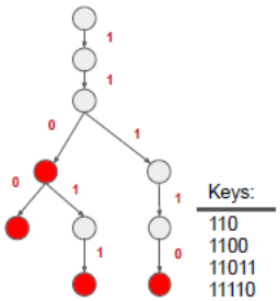


Figure 2.2: Binary Prefix Tree from [8]

In the figure above we can see a Binary Prefix Tree where binary keys are stored. As can be seen, even non-leaf nodes can identify keys.

This data structure has a benefit of memory saving for data where keys have frequently the same prefixes. For example, if we want to save all the paths of files that are present in a given folder, this can be done efficiently through Prefix Tree saving a lot of space, since the files, if present in the same folder, will surely share the same path except for the last piece. In terms of space used, all pieces of shared path will be saved in the tree once while, only the last pieces of path coinciding with the names of different files will be saved in separate leaves.

Let us say that $n$ is the number of bits required to save each piece of a path. To save paths (`folder\file`) without the use of the prefix tree we will pay:

$$n * 2 * numberoffiles\ bits$$

With prefix tree we pay, instead, only:

$$n + n * numberoffiles\ bits$$

### 2.1.2 Patricia Trie

*Prefix Tree*, however, has the weakness that if the keys share almost all the characters that make it up, you will have a tree that will branch only at the end and for the first part will consist only of nodes with only one child. In figure 2.2 for example, all the keys share the prefix '11' but to reconstruct this part of the key, in a search, anyway we will need to crawl two nodes in the tree. The problem increases when the keys have long common parts as it creates long single-branch paths in the tree.

To address this problem *Fredkin* proposed a structure called *Trie Memory* in [9]. In a Trie Memory structure, data are represented in a table in which columns are nodes and rows are branches. The number of rows depends on the number of possible branches. For instance, to represent binary data, two rows are needed. If the data are represented in a tree, the depth of the tree is equivalent to the number of columns. Every cell of this table can contain:

- reference to a child node represented by a number of a column to jump-to;
- string equivalent to a suffix of the key.

| Keys | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | | | | | (6) | AA | | |
| 1111 | 1 | (2) | (3) | (4) | (5) | | | | |
| 11110AA | 2 | | | | | (7) | | FF | |
| 11112FF | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 1111FBB | F | | | | | (8) | | | BB |

Figure 2.3: Example of Memory Trie from [8]

In figure 2.3, a Trie Memory encodes four keys composed of hexadecimal characters. First four characters for all keys are the same ('1111'). So the first four columns only refers to the next column. Then branching occurs in column '5' so comparing it to a tree at depth 5. In it we can see three

references to successive columns '6','7','8' located in as many cells at rows '0','2','F'.

The look-up starts that the input key is traversed by characters. The character at a current position is found in a row first. If this row contains a string, this path terminates and a key is found. If the cell contains a link to another node (i.e. another column), this column contains all possible branches of all encoded keys. To find a particular key, the next character from the input key is taken and the row matching this character is found. This row together with current column determines a table cell. This cell is again analysed and the algorithm recurses. In a case the cell is empty (i.e. it contains neither next column nor the string), the input key is not encoded in the table.

This data structure is effective in terms of speed because it can be represented as a two-dimensional array and then accessed by indexing this array. On the other hand we have two disadvantages:

- large memory overhead, especially when the structure is sparse;
- it groups, and thus speeds up searching, only common key suffixes (or the prefix for data saved in reverse order). Common path situations between the keys located in the middle of the keys, however, are not exploited.

These disadvantages were tackled by Morrison in his work called Practical Algorithm To Retrieve Information Coded in Alphanumeric [10], i.e. Patricia. This structure is a trie using one more optimisation. The common prefix among all keys is grouped into a single cell.

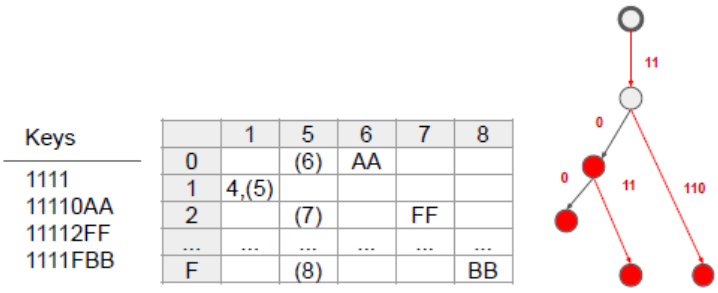


Figure 2.4: Example of Patricia Memory Trie and Patricia Trie from [8]

In the figure above we see a Patricia Trie both in array and in tree form. In the array format, in the first column is indicated the number of elements to be skipped because they are part of the common prefix ('1111' will correspond

to 4) and the next column where branching will occur (column 5). As we have already seen in the memory trie, grouping of common suffixes also takes place. The same thing happens in the Patricia trie in the form of a tree where, for example, for a hypothetical key '1110' both grouping by key prefix ('11') and suffix ('10') occurs.

### 2.1.3 Merkle Trees

*Merkle Trees* are a data structure proposed by Ralph Merkle in the book *A Digital Signature Based on a Conventional Encryption Function* [11]. In *Merkle Tree* every node in the tree contains a hash of its children. The leaves of the tree link values from the encoded datasets. Of course, cryptographic hash is used, so that the original values cannot be recovered. The purpose of this data structure is to allow the consistency of a data set to be verified without the need to exchange the data set itself. It is particularly useful in distributed environments like blockchain, where it is necessary to verify the consistency of replicated data, while the analysis of data is unfeasible because of their size. This data structure has many other advantages:

- Prevent malicious or unintentional modifications of data by providing a unique hash that identifies the data set. Another data set will only match the first one if using the same type of hashing results in the same hash encoding, otherwise the data set is different. In case a value of any node is manipulated, indeed, all hashes from this node to the root do not match and this inconsistency is revealed without the need to traverse the entire dataset.

- Participants of an exchange can also swap only sub-hashes since each subset of data is also hashed.

- Blockchain platforms involve distributed parts that do not trust each other, while needing to handle common data. In this context, Merkle trees are the ideal solution because they provide a proof of validity (the hash itself).

- The hashes are small and therefore ideal for transfer over the Internet.

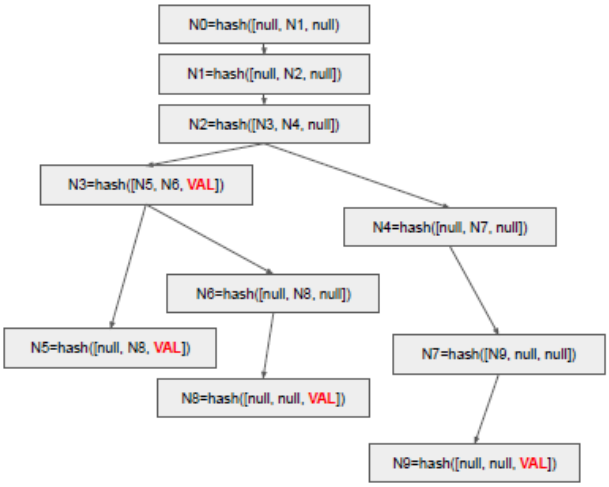


Figure 2.5: Merkle Tree from [8]

In the Merkle Tree of the 2.5 figure, data are both saved and hashed in the tree (also in non-leaf node like N3). Originally in Merkle Tree the tree is built separately with respect to location where data are stored that can be implemented with any other data structure. It can be seen that each node is made up of the hash of its children.

## 2.2 Merkle Patricia Trie

The *Merkle Patricia Trie* [6] merges main features of the:

- **Patricia Trie**: grouping of common prefixes of the set of keys saved in the structure in one node, so as to both speed up their search and pay less in terms of space used;
- **Merkle Tree**: ensures non-tampering of data saved in the structure by providing a proof of validity consisting of the hash of the root of the tree.

Merkle Patricia Trie represents not only a data structure, but also a database schema as all the keys, the data and the hashes are all stored in same tree structure. In this data structure each node can have up to 16 branches and can save hexadecimal strings. When keys have common parts, these will be saved directly in the node instead of saving only the number of bits to be skipped like in Patricia Trie. To implement the Merkle tree feature instead, any node is saved as a hash so any reference to a node is actually a reference to its hash.

This structure defines three different types of nodes with following structure:

- **Branch**: node composed of 17 items $[i_0, i_1, ..., i_{15}, value]$. This type of node is used where branching takes place, i.e. keys at certain character position starts to differ. The first 16 positions within this type of node are used precisely to allow branching up to 16 branches (from *0* to *F* hex character). The $i-position$ contains a link to a child node whenever the child exists, i.e. this position corresponds with a next character (hex *0* to *F* ) in a key. The 17th node item can store a value and it is used only if this node is terminating (the last node for the storage of a certain key and thus its value).
- **Extension**: node composed of 2 items $[path, key]$. This type of node is where the compression of common paths between keys takes place (prefix or common part in the middle of the keys). Whenever there is a part common to multiple keys, this part is stored in this type of node in the *path* item of the node. The **key** item, is used for addressing the next node.
- **Leaf**: node composed of 2 items $[path, value]$. This type of node is used to terminate the path of some key in the tree. As in the previous case, it can still adopt a compression for common paths (suffixes) that will be saved in the *path* item of the node. Also in this case the *value* item is used to save the data with the key ending in this node.

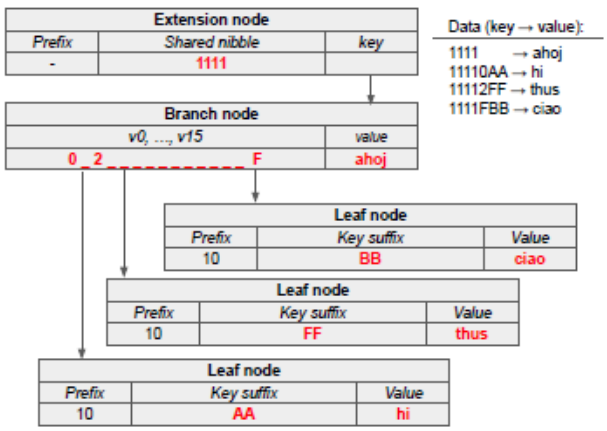


Figure 2.6: Merkle Patricia Trie from [8]

We can see from the previous image that all the values to be saved in the structure are pointed by keys with common prefix '1111'. So the first node

used is of type extension to allow the skip of this common path between the 4 keys. It then points to the next node that does the branch of the 3 keys with more than 4 characters through the hexadecimal *0−position*, *2−position* and *F−position* in the node. The branch node also stores in the item value the string *'ahoj'* with key '1111', which was skipped entirely via the previous extension node. The nodes pointed by the branch node are all leaf nodes that save in the path part the termination suffixes of the respective keys that are being encoded through the tree ('AA', 'FF' and 'BB'). In the value items, of these three nodes, are stored respectively the three strings pointed by the keys (*hi*, *thus*, *hello*). In the image, it can be seen that leaf and extension nodes include an additional field called prefix that is useful for node encoding, which will be covered in the next section.

## 2.3 Data Structures Encoding

To enable data persistence and hashing adoption in Merkle Patricia Trie there is a need to represent the nodes we analysed above as byte arrays. This is done in two steps through two types of different encoding:

1. Hex Prefix Encoding
2. Recursive Length Prefix

### 2.3.1 Hex Prefix Encoding

Hexadecimal prefix encoding is formally defined in the Appendix C of Ethereum Yellow Paper [6]. In the Merkle Patricia Trie it is used on extension and leaf type nodes and is useful for encoding data from these nodes in byte sequence. The main function of this encoding is to transform any sequence of nibbles (hexadecimal characters each one of 4-bit) into binary stream so that two (4-bit) nibbles compose one (8-bit) byte. Formally nibbles are encoded as a sequence like this:

$$\{16x_i + x_{i+1} | i \in [0, \cdots, \|x\| - 1]\} \tag{2.1}$$

Each of these sequences is further prefixed by an additional nibble (4-bit) that has two purposes:

- Act as a flag to indicate whether it is a leaf or extension node. This flag is represented by the second-lowest bit in the nibble and is set to '0' for Extension and a '1' for Leaf node;

- act as a flag to indicate whether the length of the encoded data is even or odd. This flag is represented by the lowest bit in the nibble and is set to '0' for even length and '1' for odd length. This bit is important for padding. Actually, it gives us the ability to reconstruct the original values during decoding.

The four possible possibilities are shown in the table:

| Hex | Bits | Node Type | Path Length |
|---|---|---|---|
| 0 | 0000 | Extension | even |
| 1 | 0001 | Extension | odd |
| 2 | 0010 | Leaf | even |
| 3 | 0011 | Leaf | odd |

Table 2.1: Table of possible combinations for the prefix nibble in the Hex Prefix

Since, as a rule, computers cannot store only half a byte, byte arrays resulting from Hex Prefix will still be stored with an even length. In HP, the number of encoded nibbles must always be even. To explain the problems that could occur if an even number of nibbles is not guaranteed in an HP encode, we bring an example below. Suppose that a hex value '$0 \times 10$' was encoded. In this case, it would not be possible to determine whether the original nibbles were two (with values '1' and '0') or only one (with value '1'). Instead, if we take for granted that the number of encoded nibbles is always even, '$0 \times 10$' may represent only the encoding for 2 nibbles containing '1' and '0'. Following this principle, another nibble with value '0' is inserted after the nibble prefix if the number of nibbles (including nibble prefix) is odd (the number of original nibbles encoded then will be even). This zero is a padding to extend the nibble number to an even number. If the nibble number is already even, this additional nibble is not used and the encoded path continues immediately after the first nibble. Below we show the table with the full functioning of Hex Prefix:

| Prefix | Payload | Node Type | Path Length |
|---|---|---|---|
| $[0, 0]$ | $[x_1, x_2][x_3, x_4]$ | Extension | even |
| $[1, x_1]$ | $[x_2, x_3][x_4, x_5]$ | Extension | odd |
| $[2, 0]$ | $[x_1, x_2][x_3, x_4]$ | Leaf | even |
| $[3, x_1]$ | $[x_2, x_3][x_4, x_5]$ | Leaf | odd |

Table 2.2: Table of all possible situation in the Hex Prefix

We show now two examples depicting the encoding of both an even and odd number of nibbles.

$$[6, 7, 8, 9] \rightarrow [00, 67, 89] \vee [20, 67, 89]$$

As we can see in this case starting with an even number of nibbles to encode, with the addition of the prefix nibble (which can be '2' or '0' depending on whether it is a leaf or extension node) these became odd. As explained and also seen in the table 2.2 following the nibble prefix is added an additional nibble ('0') to bring the number of elements encoded back to an even number.

$$[5, 6, 7, 8, 9] \rightarrow [15, 67, 89] \vee [35, 67, 89]$$

In the case where the starting nibbles to be encoded are odd with the addition of the nibble prefix these become even so there is no need for any additional padding nibbles. Again there are two possible solutions for the prefix nibble depending on whether the nibble to be encoded is leaf ('3') or extension ('1').

### 2.3.2 Recursive Length Prefix

After the use of Hex-Prefix encoding, *Recursive Length Prefix* is used to encode structured nodes in compressed byte arrays. Nodes in a Merkle Patricia Trie can contain multiple arrays in each node (e.g., the Branch node is a 17-entry structure, so it contains 17 values/arrays, Leaf and Extension contain two arrays). RLP is practical to flatten these arrays into a one-byte array structure for later persistence in value-key memory. Additionally, to compute a hash function, it is necessary to flatten the arrays since the hash function cannot be applied to structured forms.

Also defined in the Ethereum Yellow Paper in Appendix B [6], the RLP function basically goes to convert the sub-arrays that may potentially be

present in a node into a single long array which is the concatenation of those given as output to the function. Each sequence, in the output array, representing original sub-array is prefixed with:

- the length of the sub-array;
- a bit-mask to determine if the sub-array was originally an array or a string.

The entire sequence is then followed by the total length of the sub-arrays so every level of flattened structure will have an information regarding its total length. Each type of data to be encoded will be treated differently. They are all explained below.

- **Byte values up-to 127**:

$$RLP : b \rightarrow b \tag{2.2}$$

  Since it is a single byte, it will not be modified;

- **Strings with length up-to 55 characters**:

$$RLP : s \rightarrow [0x80 + \|s\|, s] \tag{2.3}$$

  Where $s$ is the input String and $\|s\|$ is the length of $s$.

  Example:

$$ABCD \rightarrow [0x80 + 4, A, B, C, D]$$

  With

  - $\|s\| = 4$

- **Arrays with total length up-to 55 characters**

$$RLP : S \rightarrow [0xc0 + \|ex(S)\|, ex(S)] \tag{2.4}$$

$$ex : S \rightarrow (RLP(s_i)|s_i \in S) \tag{2.5}$$

  Where $S$ is the input set of arrays, $ex(S)$ apply the RLP encoding for all the sub-arrays in the input set and $\|ex(S)\|$ is the length of $ex(S)$. Example:

$$[AB][CDE] \rightarrow [0xc0 + 7, 0x80 + 2, A, B, 0x80 + 3, C, D, E]$$

  With

- $\|ex(S)\| = 7$
- $ex([A, B]) = [0x80 + 2, A, B]$
- $ex([C, D, E]) = [0x80 + 3, C, D, E]$

- **Strings with length up-to $2^{64}$ characters**

$$RLP : s \rightarrow [0xb7 + numB(\|s\|), \|s\|, s] \tag{2.6}$$

$$numB : N \rightarrow N \tag{2.7}$$

Where $s$ is the input String, $\|s\|$ is the length of $s$ and $numB(\|s\|)$ is the number of bytes required for store $\|s\|$. For instance strings up-to $0xFF$ characters need one byte to represent their length, lengths between $0xFF$ and $0xFFFF$ need two bytes, etc.

Example:

$$A \cdots B \rightarrow [0xb7 + 2, 0x1, 0x2C, A, \cdots, B]$$

With

- $A \cdots B$ = string of 300 characters
- $numB(\|s\|) = 2$ bytes that we need for represent 300 in hex form
- $\|s\| = 300$ that in hex form is equal to $0x12C$
- $0x1$ = first byte to represent 300
- $0x2C$ = second byte to represent 300

- **Arrays with length up-to $2^{64}$ characters**

$$RLP : S \rightarrow [0xf7 + numB(\|ex(S)\|), \|ex(S)\|, S] \tag{2.8}$$

Where $numB$ and $ex$ work as explained above.

Example :

$$\begin{aligned}[A \cdots B][ABCD] \rightarrow [&0xf7 + 2, 0x1, 0x34,\\ &0xb7 + 2, 0x1, 0x2C, A, \cdots, B,\\ &0x80 + 4, A, B, C, D]\end{aligned}$$

With

- $A \cdots B$ = array of 300 characters
- $[ABCD]$ = array of 4 characters

- $numB(\|ex(S)\|) = 2$ bytes that we need for represent $300 + 4$ in hex form
- $\|ex(S)\| = 300 + 4$ that in hex form is equal to $0x134$
- $0x1$ = first byte to represent $300 + 4$
- $0x34$ = second byte to represent $300 + 4$

- **Payloads with length higher than $2^{64}$**

  Neither strings nor arrays of this size can be encoded.

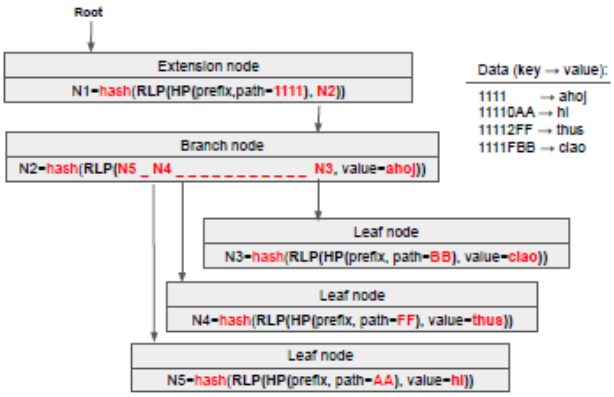


Figure 2.7: Merkle Patricia Trie Encoded from [8]

As you can see from the 2.7 figure on the extension and leaf nodes, HP encoding was used first, while, then, RLP encoding was used on all nodes to transform them into byte arrays. At this point an asymmetric hash function can be used on them. After that, parent nodes refers to its children via this hash.

### 2.3.3 Persistence in Key-Value Storage

Once the entire structure has been transformed as a byte array and subjected to an asymmetric hash function we would like to store it in a database for data persistence. Having hashed the nodes, it is not possible to get the original data from the nodes so we cannot save just the hashes. Instead, it will be necessary to structure a key-value database in which each node is represented by its key (the hash computed on the node's byte array) that points to the value represented by the byte array saved in plain text. The

function hash used by Ethereum is called Keccak [12], which is an original implementation of SHA3 available before SHA3 became official.

$$key \equiv (keccak(RLP(node))) \rightarrow value \equiv RLP(node) \tag{2.9}$$

Where the definition of *node* differs depending on the node type:

- **Extension** :
$$node \equiv [HP(prefix + path), key]$$
Where *key* means the key that identifies its child;
- **Leaf** :
$$node \equiv [HP(prefix + path), value]$$
- **Branch** :
$$node \equiv [branches, value]$$
Where $branch = (key_i | 0 \leq i \leq 15)$ and $key_i$ is a hashed key of a child node if the branch exists.

## 2.4 Blockchain, Block and Transaction

Blockchain technology is part of the larger family called *Distributed Ledger*. Distributed Ledgers are systems in which all nodes in a network have the same copy of a database that can be read and modified independently by individual nodes. Whereas, in so-called *Distributed Databases*, all nodes that own a copy of the database can consult it, but must go through a central entity (or multiple validating entities) to modify its data.

In Distributed Ledger systems, changes to the registry are governed by *consensus algorithms*. Such algorithms allow consensus to be reached among the various versions of the ledger, despite the fact that they are updated independently by different participants in the network. An in-depth discussion of the consensus algorithm used in Ethereum (*Proof of Work consensus*) will be covered later in the section called *Proof of Work consensus*. In addition to consensus algorithms, to maintain the security and immutability of the ledger, Distributed Ledger and Blockchain in particular, make extensive use of cryptography. Various examples of Distributed Ledger differ from the different ways in which the ledger is structured. In Blockchain, for example, the ledger is structured as a chain of blocks concatenated via cryptography hashes (such as in Bitcoin or Ethereum platforms).

Blocks are used by various users to update a blockchain. The main purpose of each block is to execute transactions. Stakeholders participating in

a blockchain ecosystem send new transactions, which usually represent their desire to transfer funds between user accounts (sender and receiver). Typically, the transaction causes that sender account's balance is deducted while the receiver account's balance is increased, i.e. the funds flow from the sender to the receiver. Once validated, the transactions are enclosed in a block, the block is attached to the blockchain via the hash and it becomes part of the chain forever.

Moreover, the block in Ethereum contains a list of *ommers*. This list consists of headers of blocks that were successfully mined, but they eventually did not became part of the mainline. In some cases, actually, in Ethereum blockchain is possible to have two branches at the same time. One of them is later dropped to have in the end only one main chain. This eventuality is due to how the process of validating the various transactions and saving them within the blocks is designed and takes place. This process is called *mining* and is carried out by a group of stakeholders called *miners* responsible for validating the network, collects these transactions and packages them into blocks. To do this, they have to verify the transactions, calculate the hash of a parent block and perform (artificially) resource-intensive calculations to prove that they invest considerable effort in the mining activity. For this work they get rewards and in the end the block became part of the blockchain.

However, this process can be carried out by different stakeholders at the same time, and thus it can happen that several blocks with the same transactions in them are produced at the same time. These redundant blocks will create two branches in the blockchain tail. One of the two will be eliminated through the consensus protocol that will decide the leading path. In short, the consensus protocol having to choose between 2 different branches will prefer the one in which more computation power has been invested. Redundant blocks discarded by the consensus protocol, are not thrown, but become *ommers* for the leading block, and those who mined them still receive a lower reward.

So, a block $B$ contains: a header $H$, transactions $T$ and ommers also referred as uncles $U$.

$$B = (H, T, U) \tag{2.10}$$

The part that is used to identify a block and contains all its meta data is the header $H$, which consists of this list of fields:

- **parentHash**: The Keccak 256 bit hash of the parent block. This field connects blocks in a chain;
- **ommersHash**: The Keccak 256 bit hash of list of ommers;

- **beneficiary**: An account address of a user who mined this block and receives reward;
- **stateRoot**: The Keccak 256 bit hash of a root of the World State Trie;
- **receiptsRoot**: The Keccak 256 bit hash of a root of the Transaction Receipt Trie;
- **transactionsRoot**: The Keccak 256 bit hash of a root of the Transaction Trie;
- **logsBloom**: A filter relating logs hashes with the log records;
- **difficulty**: A scalar value corresponding to an effort that has to be undertaken to mine this block;
- **number**: A scalar value equivalent to an ordinal number of this block. Every new block in the chain gets a number increased by one;
- **gasLimit**: A scalar value containing an accumulated gas limit required to process all transactions in this block;
- **gasUsed**: A scalar value containing an accumulated real consumed gas to process all transactions in this block;
- **extraData**: A free form, 32 bytes or less long byte array, containing any additional data;
- **mixHash**: The Keccak 256 bit hash confirming that a sufficient computation has been spent on mining this block;
- **nonce**: A 64 bit value. This value combined with mixHash also proves that the computation has been spend for mining this block.

### 2.4.1 Proof-of-Work consensus

The main purpose of Proof-of-work is to enforce a certain computing power for block validation, regardless of the number of nodes (i.e., computers) added to the network. This ensures the security of the network by preventing those with prevailing computing power from rule the network. Ethereum uses the *Ethash* protocol [13] for mining and validating the mining job. This protocol derives a *nonce* that is a cryptographic hash proportional to *difficulty*. The *difficulty* is adjusted over time to ensure that the overall time spent mining blocks remains approximately the same as the computational power

invested in mining increases. Proof-of-Work consensus is based on solving a cryptographic puzzle that works as follows.

1. Mainers are provided with a set of encrypted hashes;
2. they randomly select a subset of hashes from this set;
3. this subset is hashed and produces *mixHash*.
4. if ($mixHash < nonce$)
   - than the computation is successful
   - else restart from step 2

The first miner who finds the solution that meets the required condition is the winner, it is rewarded and the block becomes part of the chain.

## 2.5 Data Representation

To explain how the Ethereum blockchain and all the data stored in it are physically linked together using the data structures previously analysed, we show below a schema of Ethereum's structure.

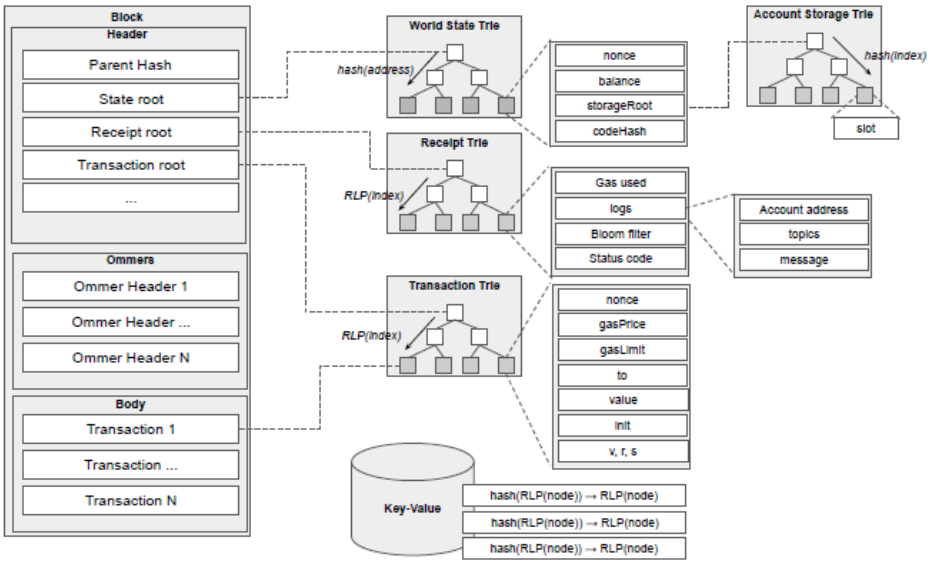


Figure 2.8: Ethereum Data Structures from [8]

As we can see from the figure, each block header points to several Merkle Patricia Trie that we will specify below. These are:

- **Transaction Trie** (pointed by **transactionsRoot** field in a header)
- **Receipt Trie** (pointed by **receiptsRoot** field in a header)
- **World State Trie** (pointed by **stateRoot** field in a header)

Among these three trie, the Transactions Trie and Receipts Trie save the execution of each transaction, which then becomes unchangeable after the transaction is executed. It means that all executed transactions are recorded forever and cannot be undone. In contrast, the World State Trie is continuously updated since it is responsible for keeping track of the most recent state of blockchain. The World State Trie is updated to reflect either account balance changes or Smart Contract creation, while the record in Transaction and Receipt Tries become verbatim once the transaction is executed.

Again from the figure we can see that separately from the blocks that make up the chain there is also a key-value database that saves the nodes of the trie according to the formula 2.9 seen earlier.

### 2.5.1 Transactions Trie

Being a Merkle Patricia Trie, the transactions trie saves both information and keys to which the information can be found. In this case, the keys, mapped as the path at which the information is findable, consist of the transaction *indexes*. The values at the end of these paths are made up of the *transactions* themselves. Both the transaction indexes and the transactions are encoded using RLP. So, in the end, we have a key-value pairs of this form stored in the trie:

$$RLP(index) \rightarrow RLP(transaction) \tag{2.11}$$

Each transaction contains these fields:

- **Nonce**: an ordinal number of a transaction. For every new transaction submitted by the same sender, the nonce is increased. This value allows for tracking order transactions and prevents sending the same transaction twice;
- **gasPrice**: A value indicating current price of gas;
- **gasLimit**: A value indicating maximal amount of gas the sender is able to spend on executing this transaction;
- **to**: Address of an account to receive funds, or zero for contract creation;
- **value**: Amount of funds to transfer between external accounts, or initial balance of an account for a new contract;

- **init**: EVM initialisation code for new contract creation;
- **data**: Input data for a message call together with the message (i.e. a method) signature. A message call is a mechanism to enable inter-contract communication, such as calling other contracts or sending Ethereum to non-contract accounts;
- **v, r, s**: Values encoding signature of a sender used to digitally sign a transaction. Values $r$ and $s$ are outputs of an ECDSA signature algorithm [14]. ECDSA is a symmetric function based on elliptic curves, which generates the signature from private and public key pairs. Value $v$ is the recovery id that is, a one-byte value, which allows the public key to be retrieved from the signature.

It should be specified, however, that the fields listed above, depending on the type of transaction performed, may be partly empty and partly filled. This happens because in Ethereum there is the possibility of using transactions for three different purposes:

- to transfer funds between accounts (simple transaction);
- to create a Smart Contract;
- to send Message calls.

| Field | *to* | *data* | *init* |
|---|---|---|---|
| Exchange funds | recipient address | | |
| Contract creation | | | byte-code + data |
| Message call | contract address | signature + data | |

Table 2.3: Table of fields per each type of transaction

In the table 2.3 we can see the fields needed for each type of transaction. Specifically , for contract creation, only the field *init* is filled while *to* and *data* fields are empty. For message call transaction, is filled the *to* field, with the contract address, and the *data* field. Finally, for transactions that transfer funds both *init* and *data* are empty while the *to* is filled with the recipient address.

Each transaction execution has a cost called *Transaction Fee*. This cost is determined through the *gasPrice* and *gasLimit* fields. These in turn should in principle be adjusted according to changes in costs in the real market for goods such as electricity and the price of hardware components. When an Ethereum virtual machine executes a smart contract, it charges a certain amount of gas per instruction and per storage space used. This fee is hard-coded, but has been updated in the history of Ethereum through so-called *hard forks*. Hard fork is a radical change to some protocols of a blockchain network. The latter can involve either an increase or decrease in the cost for a particular transaction in a contract (`SSLOD, SSTORE`), but also a change that can result in the validation of blocks or transactions that previously would have been invalid. In the last case the hard fork may result in a complete split of one blockchain into two different ones.

The *gasPrice* is set by the transaction submitter. When the miner executes the transaction, the gas spent to carry out the transaction, is multiplied by the *gasPrice* and it becomes a reward send to the miner's account.

Gas price naturally fluctuates with changes in financial markets. Add to this the fact that smart contract creators want to spend as little as possible to execute their contract. In contrast, miners tend to choose to work on contracts that offer better *gasPrice*. In the end, thus, they will have to find a meeting point. This situation will lead developers to create more optimised contracts to waste as little gas as possible, and miners by receiving compensation for their work will be encouraged to keep the chain alive. In this way, a decentralised system in which untrusted third parties can enter and leave at will, such as blockchain networks, is self-regulating.

Instead, the *gasLimit* is set by the transaction creator and expresses maximal gas, a sender is willing to pay, for executing his or her transaction. This limit is a kind of lifesaver, preventing the infinite execution of code that would drain all the sender's funds. The *gasLimit* is very important for the sender since the execution of a transaction is interrupted if it reaches the *gasLimit* prematurely. All the gas already consumed, however, is not returned to the sender since the miner still spent computational energy to execute the transaction up to the limit. In this case, the transaction initiated by the sender will not be accepted since its execution has not been completed, but he will still have to pay for the partial execution of the transaction.

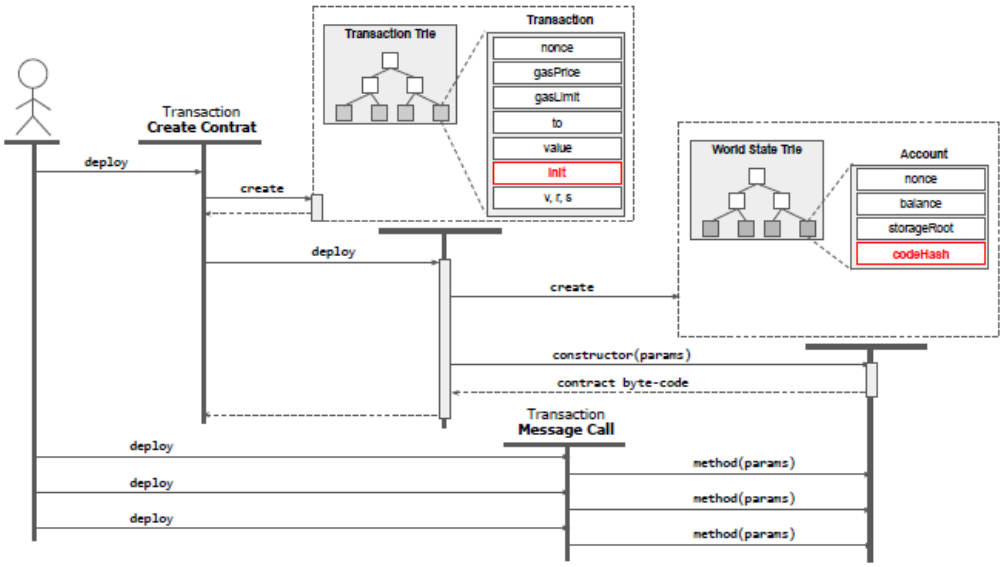


Figure 2.9: Smart Contract Creation and Invocation from [8]

Let us now turn to the type of transaction involved in the creation of a contract. As we saw earlier, the deployment of a contract on a blockchain is done through transaction.

As we can see in figure 2.9, a user visualised as a human actor, in the sequence diagram deploys the transaction to create the contract which, will then be executed. We note that the transaction is not executed immediately, actually, it must first be validated by a miner and in order to make the transaction becomes part of the blockchain, all participating nodes will sooner or later synchronise, i.e. execute the transaction.

Once executed, the transaction creates a record in the *Transaction Trie* with all the fields related to the transaction. For contract creation, the field *init* in this transaction is filled with an EVM byte-code that generates a new contract. This bytecode is generated through a smart contract compiler very often through Solidity language [15]. The *to* field in the contract creation transaction is blank because part of contract deployment is the creation of a new account. Therefore, the value of the *to* field is still unknown.

Once the deployment is finished, actually a new account for the contract is created. In image 2.9 you can actually see the addition of a new actor consisting of a new record in the World State Trie for the contract account. This new account retains in the *codeHash* field the bytecode mentioned earlier. Connected to this account is also a storage space consisting of a *Storage Merkle Trie* pointed by the *storageRoot* field of the account. Actually, at the time of creation, a contract may already contain an initial state consisting

of global variables that will, therefore, be saved in this storage tree. When a new contract is created it may also already have initial funds, which can be found in the *balance* field of the new account linked to the contract.

As mentioned, a transaction can also be used as a *message call*. Once a contract deployment has occurred, it can be executed many times. Executing a contract means precisely calling and executing the public methods it offers. It is exactly in this area that the type of transaction called *message call* is used. As we know from previous explanations it has a non-empty *data* field. This field wraps the signature of a method offered by the contract followed by its parameters in these forms:

- **method signature** is stored in a hashed form using keccak [16].
- **input parameters** are encoded in a standardised binary format called ABI format [17]. This encoding allows the creation of a single binary sequence consisting of the method signature and its input data.

This concept is similar to RPC (Remote Procedure Call) or RMI (Remote Method Interface), already well known from other languages such as Java [18]. In message calls, in addition to the *date* field, there is also a non-empty *to* field which contains the account address of the contract offering the methods we are trying to invoke remotely. The account address in the *to* field is used to fetch the *codeHash* field of the account that contains the byte code needed to execute the method we are interested in.

The methods offered by one contract can also be invoked by other contracts within their code. These transactions called internal transactions work like regular method calls. However, they have the particularity that they are not saved on the blockchain and do not have a *gasLimit*. Actually, the gas spent on their execution is transitively charged to the caller of the contract using the internal transaction and so on until the last caller. This gives the possibility of creating, through contracts, libraries as well, which are not priced themselves, but the cost in gas of the methods they offer will be considered only if they are used within another contract.

### 2.5.2 Receipts Trie

As also mentioned relative to image 2.9, once a transaction is submitted, it is not executed directly. It is actually first executed by a miner who through his work, may be able to add it to a block. Then, through an ad-hoc sequence, it is synchronised by all nodes participating in the blockchain network. Because of this process a user who submits a transaction, would not have its result immediately available. To overcome this problem, the results of transactions

executed by miners but not yet added to the blockchain will be saved in a so-called *Transaction Receipt Trie*.

This trie maintains an entry for each transaction that contains at least one result code indicating whether or not the transaction was successfully executed. Receipts are also needed in case a user is interested in knowing when a transaction they requested is executed over time. For this purpose, Ethereum offers an API for registering to the execution event of a transaction. Using it, then, a user can wait in anticipation of the transaction execution event by obtaining the receipt as soon as it is executed. Similar to how the *Transaction Trie* did for transactions, the *Transaction Receipt Trie* internally encodes the *index* of a transaction $T$ as a path to reach the record containing the receipt $R$ of $T$. In trie optics, the *index* of $T$ is used as the key to reach the value formed by $R$. They thus form a key-value pair again, encoded via the RLP-algorithm, as for all the trie previously explained.

$$RLP(T - Index) \rightarrow RLP(R) \tag{2.12}$$

The fields contained in a receipt are as follows:

- **Gas used**: the accumulation of gas used by already executed transactions, contained in the same block of the current transaction, plus the gas used by the execution of the current transaction;
- **Logs**: set of log messages generated by smart contracts for more detailed debugging of transaction execution. Each log in turn contains a tuple formed by these fields:
  - **address** of the contract account that triggered the log, i.e., that executed this Smart Contract;
  - **event topics**, consisting of a sequence of 32 bytes;
  - **log message** itself also formed by a sequence of bytes.

  A log message can contain any information or type of data. Then, each of them can be assigned to a specific topic so that a client interested only in it can receive only logs concerning that topic. From figure 2.10 we can see the process of creating a log. The steps depicted in the UML sequence diagram are these:

  1. an actor representing a user submits a transaction that executes a smart contract. It does this, for example, to execute a method of another contract via a message call type transaction. This transaction will then have in the *to* field the account of the contract

whose method you want to execute. From the *codeHash* field of this account in the *World State Trie* we are able to find the bytecode of the contract we are interested in;

2. the contract is then executed;
3. let us take as an example the Solidity language, in which the `emit` keyword is used to order the propagation of a log. If the method of the executed contract contains the `emit` keyword, the message is propagated in the log entries.

- **Bloom filter**: contains hashes of log entries to speed-up their searching. A Bloom's filter is a data structure designed by Burton Bloom [19] to tell, in a quick and memory-efficient way, whether an element is in a set. The price to pay for this efficiency is that a Bloom's filter is a probabilistic data structure, i.e. it tells us that the element either definitely is not in the set or may be in the set;
- **Status code**: A numeric code that identifies the result of a transaction.

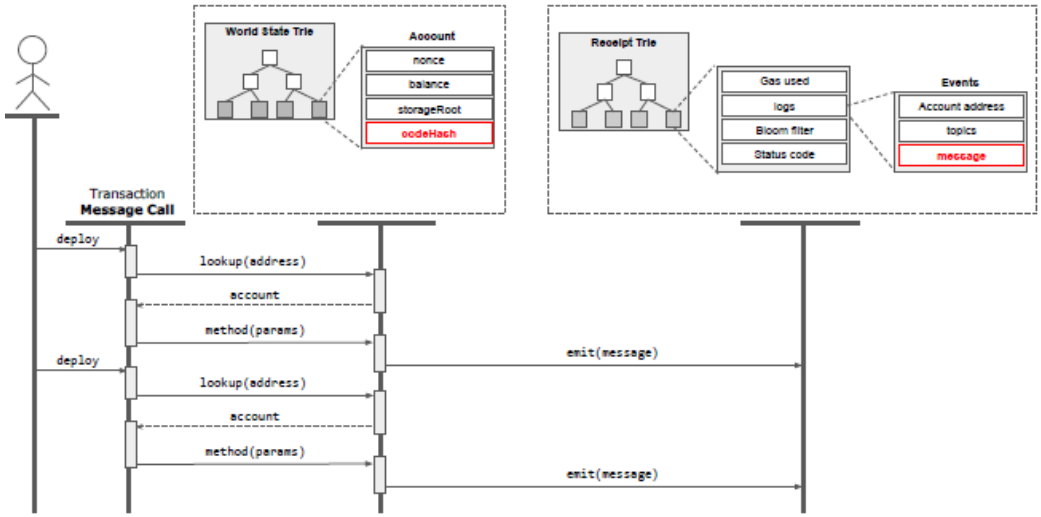


Figure 2.10: Log Event Updates Transaction Receipt from [8]

### 2.5.3 World State Trie

The *World State Trie* is, as mentioned earlier, an ever-changing data structure that always goes to depict the latest state of the blockchain. As in the other trie, here, we also have a structure that maps both keys and values within it. Again, the path to which the value can be found is the key itself.

Within this trie, the keys consist of the account *addresses* while the values are the *accounts* themselves. Let us now explore the form of the addresses and accounts:

- An **address** of an account is a 160-bit (20 bytes) identifiers, which is created as first 20 bytes of a public key from the user signature. Within the *World State Trie*, addresses are encoded with keccak [12] hashes.
- Regarding **accounts**, however, we can say that Ethereum provides two types of accounts:
  - **Externally Owned Account**: accounts for physical users. It is provided for the purpose of maintaining the user funds as an account with a real bank;
  - **Contract Account**: created for a contract just after its deployment. It is used to maintain all the data needed to create the contract.

  As with transactions, these two types of accounts are differentiated by the empty and non-empty fields in the account itself.

As with the previous tries we show the function that, from the key leads to the value in this trie:

$$Keccak(address) \rightarrow RLP(account) \tag{2.13}$$

An address consists of these fields:

- **nonce**: A scalar number.
  - for externally owned accounts it depicts the number of transactions sent from this address;
  - for contract accounts it represents the number of times the contract was created;
- **balance**: A scalar number representing the funds that this account has available.
- **storageRoot**: A 256-bit hash.
  - for externally owned accounts this field is empty;
  - for contract accounts contains the root hash of the account storage trie;

- **codeHash**: Hash of the EVM bytecode.
  - for externally owned accounts this field is empty;
  - for contract accounts maintains a hash that is then used as a key in an underlying database to point to the EVM bytecode of the contract.

The status of this structure is changed through the executions of new transactions. Of course, depending on the type of account, it accepts different types of transactions. For example, an externally owned account does not provide for receiving message call type transactions, which, instead, are expected for contract accounts. Returning to the concept of modifiability, for example, a transaction to an external owned account will change its balance field. Looking instead at a transaction to a smart contract account, it will probably go to modify its internal state saved in the storage state trie. This suggests that as an account changes very often the associated storage state trie, when intended, is also changing.

### 2.5.4 Storage Trie

As mentioned earlier, a *Storage Trie* is a structure linked to a smart contract account via its *StorageRoot* field. This can be directly editable by the bytecode of the contract account via the `SLOAD` and `SSTORE` instructions. Being a trie, it too involves mapping key-value pairs. In this case the key is an *index* that will act as a path to reach a structure called *slot* that may contain one or more variables that constitute the internal state of the contract. The *slot* is wide 32 bytes and its content depends on a data type of particular variable, furthermore detailed can be found in Solidity compiler documentation [20] and a more colloquial treatment can be viewed at the blog [21]. How the assignment of variables within slots takes place is discussed later in this section. As with the indexes in the other trie, they are also subjected to hashing. Below we show how these pairs are encoded in the trie.

$$Keccak(index) \rightarrow RLP(slot) \tag{2.14}$$

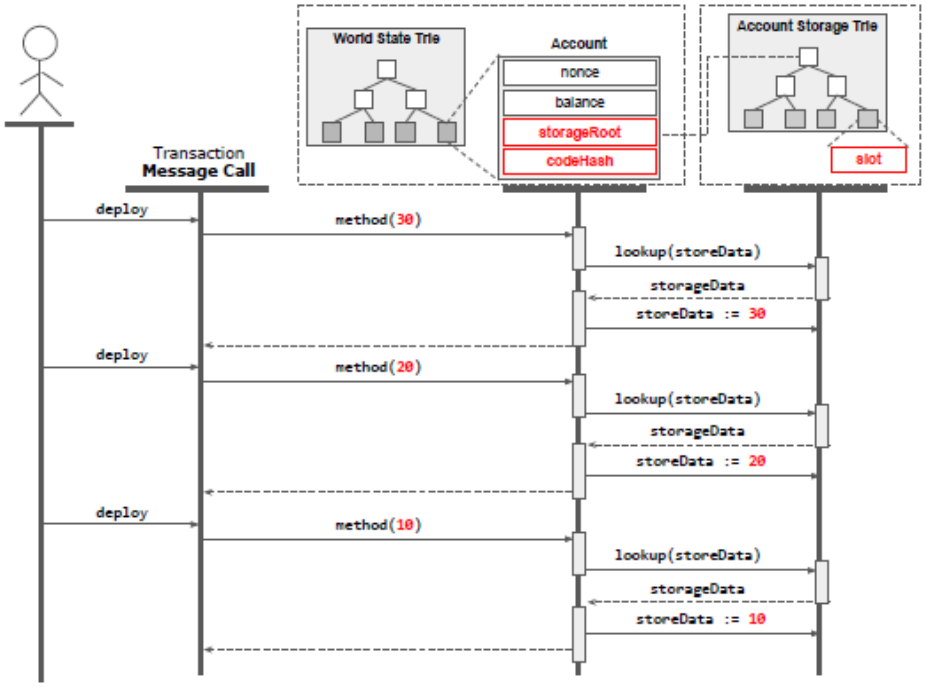


Figure 2.11: Storage Trie Update from [8]

To explain the figure 2.11 we take the following statements for granted:

- a user, an actor in our sequence diagram, has already submitted the transaction for the execution of the smart contract (i.e., call a method of the contract via message call);
- the date field in the transaction (message call) contains the signature of a function named `method`;
- these function has been called three times with values 30, 20, and 10. In the respective transactions (message calls) these values will be concatenated to the signature of the `method` function always within the *data* field;
- the `method` function that, the user (actor) wants to execute, has a global variable called `storedData`;
- the `method` function is a setter for the global variable `storeData` i.e., it is a function that gives the ability to modify the variable `storeData`.

Following these observations, we can say that in the image 2.11 three times this sequence of events occurred:

1. the account placed in the to field of the transaction is identified in the *World State Trie*;

2. the contract searched via the *codeHash* field of the previously found account is executed (i.e., the `method` function of the contract is executed);

3. The account storage trie connected via *storageRoot* is updated (Since the `method` function goes to change the internal state of the contract consisting of the `storedData` variable).

4. The account storage trie is sequentially modified as the transactions are processed.

Turning to the *slot* conformation we first say that the data format valid throughout the blockchain is defined in *Solidity*. This suggests that, if you want to use another data format through perhaps, another compiler, that format will not be compatible with the blockchain. This will be true even if the solidity compiler decides to change its way in which the data are stored. Starting from this premise, let's explore how the current version of solidity organises variables into slots.

- **Statically Sized Variables**: multiple items are grouped in one slot up to total of 32 bytes. The first concatenated item, whose bytes exceed the slot threshold, will be moved completely to another slot. Structures expected by solidity and arrays always start a new slot and occupy it completely. A progressive index is assigned to the slots by the compiler. These start at index `0x0` and go up.


Figure 2.12: Statically Sized Variables from [8]

The previous image 2.12 depicts two cases of filling two different slots. In the first, three variables (`int128`, `int8` and one `bool`) are stored in one slot. In the second, only one variable `int256` is inserted.

- **Maps**: First, let us recall what a map is. It is a data structure where many key-value pairs are stored. In which, the *key* is an identifier for a certain type of data, and the *value* is the content that is identified or stored. As in many other data structures, the keys are unique, so no

duplicate keys are possible. Starting from this definition, we can say that saving a map data structure within slots exploits this property. As a first step, the first available slot will be selected. Assuming that this slot is pointed by the index *index*. For every key in the map, the key is concatenated (+) with this slot *index* and the hash of the trie key is calculated in this way:

$$index_{val} = keccak(index + key) \tag{2.15}$$

These $index_{val}$ point to slots that contain the map values

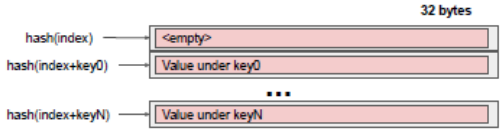


Figure 2.13: Map Variables from [8]

In the figure above 2.13 we see how the first slot pointed by the index *index* is empty while the subsequent slots pointed by the $index_{val}$ of the form $index + key_i$ are pointing to the respective values pointed by the $key_i$ in the map.

- **Dynamic Arrays**: As in the previous case, we quickly define an array. An array is a structure that provides a fixed or variable *size* and depending on the latter gives the possibility to include *size* elements inside it. Thus, an element may be included in the structure at position $i$ where $i \in [0, size)$. Going back to the main discussion regarding the assignment of a dynamic array to slots let us say that the first available index *index* pointing to the next empty slot will be selected. Let us then say that the index of the trie pointing to the array elements will be named $index_i$ and will be calculated in this way:

$$index_i = keccak(index + i) \tag{2.16}$$

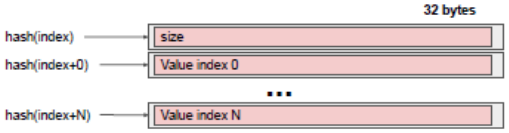


Figure 2.14: Dynamic Arrays Variable from [8]

From the figure 2.14 we can see that first of all, unlike the map case, the slot pointed by *index* is not empty but in it is contained the *size* of the array. Then consecutively $index_i$ points to a slot with the value from array index $i$. The array have an additional benefit over the map since it can pack values if they fit more variables into one slot.

- **Byte Arrays and String**: When these types are shorter than 32 bytes, they are stored in a slot so that one byte contains the size and 31 bytes are used for data. For longer strings or arrays, they are stored in the same way as dynamic arrays. One slot contains the length and subsequent slots the payload. In the figure 2.15 we can see an example of a situation where the byte array or string has size less than 32 bytes.


Figure 2.15: Byte Array and String Variable from [8]

### 2.5.5 Authenticated Storage

One of the most important concepts for my thesis work concerns this property of authenticated storage. This property is one of the many reasons why there is a widely used, within the Ethereum blockchain, of the *Merkle Patricia Trie*. They, in fact, inherit from *Merkle Tree* the property of authenticating and validating the data within the trie by the hash coming with the data itself. We show the principle through the following figure.

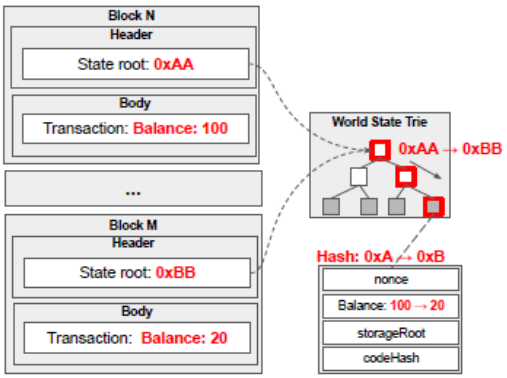


Figure 2.16: Authenticated Storage from [8]

From the figure 2.16 we can see that first we have a block N that contains within it a transaction that brings the balance of a certain account to 100.

The state of the balance of this account, as we know from the topics discussed above is stored in the *World State Trie* pointed by block N. Since each piece of data come with his own hash, we see from the figure 2.16 that, this transaction caused the hash of the account to change in *0xA*. This change, in cascading, also leads to the modification of the hash of the root of the *World State Trie* in *0xAA*.

After some time, an additional block M is added to the blockchain which contains a transaction that brings the balance of the account considered before from 100 to 20. This leads to all the cascading changes seen in the case of block N. In this case the hash of the account will become *0xB* while that of the *World State Trie* will become *0xBB*. This new hash is stored in the header of the M block, and being more up-to-date than the one in the N block, the latter will become obsolete. The result is that both the data structure and the blockchain are bound to these hashes.

Without this mechanism, a malicious user, changing the data source, might try to prove that he has more funds than he really has. Another possible scam case without this property is one in which, after a transfer of funds for some good, the malicious user wants to get back into possession of the spent funds so as to have both the asset purchased and the funds in his possession before the purchase. Thanks to the principle of authenticated storage, it will be easy to discover the malicious user's cheating since, the hash that come with his data will surely not match that of the updated *World State Trie*.

# Chapter 3

# Related Work

In this chapter we present some of the works that are closest to the solution proposed in this thesis. These are all *Layer 2* solution.

To understand what these *Layer 2* solutions deal with and the reason why they exist, we need to take a few steps back and talk about the weaknesses of a blockchain system. Anyone who approaches the world of blockchain will soon realise that despite the fact that, on paper it is a system with great potential that offers security and integrity, but, on the other hand, lending itself to a very large weakness consisting of scalability. Scalability refers to the ability of a system to maintain acceptable performance as the computational workload to be performed increases. For example, in *Bitcoin*, the increasing number of transactions makes Proof of Work ($PoW$) execution very impractical in terms of execution time and computational resource usage. This leads to a large validation time per transaction, making the user experience very negative.

The architecture of a blockchain being distributed, like many other systems, faces a problem very similar to the so-called CAP *(Consistency, Availability, Partition fault tolerance)* theorem [22]. To explain this problem colloquially, we can compare it to the case when someone is cold and want to cover his whole body (shoulders, belly, feet) with a blanket that is, however, not long enough to cover the whole body. The blanket is actually long enough to cover only two out of three parts of the body. Similarly, the CAP theorem states that in a distributed system only two properties among Consistency, Availability and Partition fault tolerance can be guaranteed at the same time.

Going back to blockchain, a very similar problem called *blockchain trilemma* is present in this area and deals with the three properties: *decentralisation, security* and *scalability*. This is easily understood, since a fully decentralised blockchain is secure but does not scale very well (previous example of Bitcoin), a blockchain that scales well must have a centralised point

to achieve an acceptable level of security and a decentralised blockchain that scales well cannot be secure. A trade-of will must then be found among these three properties to find an appropriate balance to provide a good user experience.

Another key piece of information we need, to talk about *Layer 2* solutions is to understand how a blockchain technologies, nowadays, can be categorised into a stack of layers. We can recognise 4 layers (from Layer 0 to Layer 3):

- **Layer 0**: Blockchain in itself is called layer zero. The components required to make blockchain real are the internet, hardware, and many other connections. Layer zero blockchain is the initial stage of blockchain that allows various networks to function, such as Bitcoin and Ethereum. It also provides blockchain with a facility of cross-chain interoperability communication from top to different layers. In simple terms Layer 0 provides the underlying infrastructure for blockchain;

- **Layer 1**: also called "implementation layer", is an evolution of layer 0 and use its offered services. Therefore, any changes and problems that arise in the layer 0 protocols will also affect layer 1. Examples of layer 1 blockchains are Bitcoin and Ethereum;

- **Layer 2**: is a collective term to describe a specific set of solutions that allow for the scaling previous layer blockchains. Layer 2 solutions are overlapping networks that lie on top of the Layer 1 extending it and inheriting its security properties.

- **Layer 3**: also referred as the "application layer". The main task of this layer is to host the decentralised networks applications (Dapps) and many other protocols that enable other apps.

Looking at Ethereum, as we know, it is a blockchain that offers complete decentralisation and security. Research in this area has therefore focused on increasing performance regarding the scalability of this blockchain. Nowadays, three main categories of solutions for increasing scalability have emerged. They are divided according to the level of the blockchain where these solutions act to increase scalability.

- **Layer 0**: This class of approaches attempts to improve the scalability of the blockchain optimising the information propagation in the main blockchain network. This first approach turned out to be the least interesting one to explore in research.

- **Layer 1**: These types of solutions act directly on Layer 1 blockchains and their components, such as the consensus algorithm, block structures and chain data structures. This type of approach therefore tends to modify standard components of a blockchain leading to hard forks as well.

- **Layer 2**: With Layer 2 solutions, the goal is to reduce the computational load on the main blockchain eliminating some interactions on it performing some transactions or some complex calculations in an off-chain environment, outside the main blockchain. As a result on the main chain only a commit of some data regarding these transactions made off-chain occurs. The latter are required to verify that off-chain transactions are compliant with regulatory requirements and to maintain the property of immutability regarding these transactions. The data entered into the main chain can be of different types and range from summary of the off-chain computation to some kind of proof related to the off-chain activity.

Below we can see a figure showing the Layer 2 solutions:

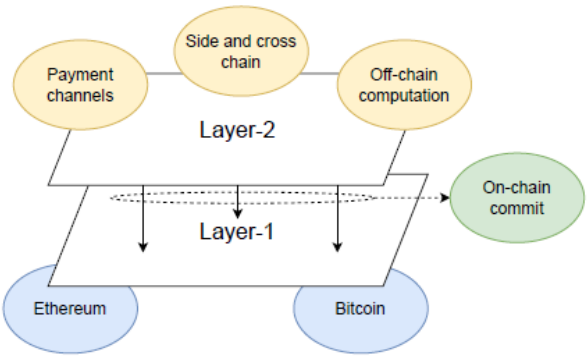


Figure 3.1: Layer 2 solutions

We focus on the last type of solutions that are the ones considered most promising by the developer community. In this solution, a crucial point is interoperability, since you want to ensure that the whole system is maintained on a solid protocol that guarantees fast (thus optimised) and integer data exchange.

For this purpose, 5 subcategories were developed that try to act on different aspects of interoperability. The discussion will now be divided into sections related to these subcategories, in which, we will present related im-

plementations. Below we show a summary table of these solutions, with the relative main implementation.

| **Solution** | **Main implementation** |
|---|---|
| Payment channels | Lightning network |
| Proof systems | Merkle proofs |
| Cross-chain | Cosmos, Polkadot |
| Side-chains | Polygon |
| Rollups | ZoKrates, Arbitrum |

Table 3.1: Layer 2 solutions classification

## 3.1 Payment Channels: Lightning network

Payment channels are layer 2 solutions for scaling a main blockchain. With only 2 transactions, an opening transaction and a closing one on the main blockchain, a user can open a payment channel where the user can go to perform different off-chain transactions.

### 3.1.1 LightningNetwork

The Lightning network[23] is a network of payment channels built on Bitcoin's blockchain [24]. This system aims to allow users to freely exchange bitcoin in dedicated and secure channels, reducing the number of transactions on the main blockchain.

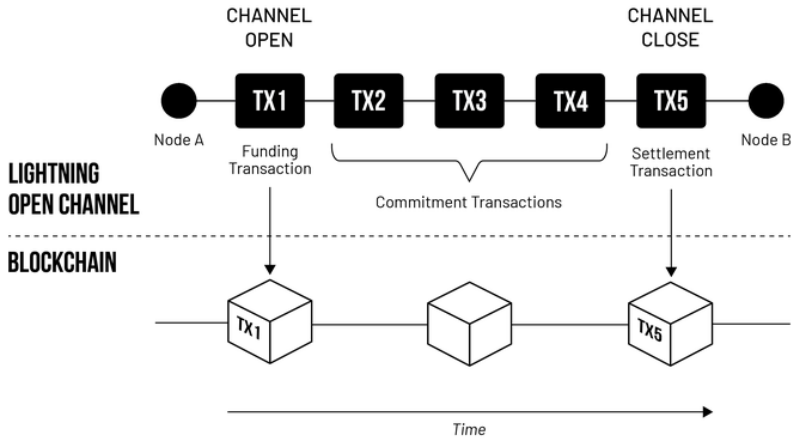


Figure 3.2: Example of Lightning Payment Channel [25]

As shown in the Figure 3.2, this solution works in three main steps:

1. users setup an off-chain channel by registering an open transaction called *Funding Transaction*;
2. they perform one or more *Commitment Transactions* on the off-chain payment channel;
3. the channel, in the end, is closed by committing on the main chain a summary of the transactions made in the off-chain channel. This summary transaction is called *Settlement Transaction*.

As can be easily seen, this solution is easily attributable to a Layer 2 solution since it proposes a method to decrease the workload on the main chain.

## 3.2 Proof systems: Merkle proofs

As we mentioned, *Layer 2* solutions are also concerned with providing methods to attest some data properties. These properties can be the presence, the integrity or the consistency in time of the data when you want to make a commitment to the main chain. Proof Systems are concerned with addressing precisely this need. One of the most popular solutions in this area, which has also been used in the thesis project is *Merkle Proofs*.

### 3.2.1 Merkle Proofs

Easily understandable from the name, *Merkle Proofs* are generated by using a *Merkle Tree* and exploiting its properties to prove that a given node exists,

has not been tampered and is consistent in two time moments. As we know *Merkle Tree* are structured in such a way that, starting from the leaf nodes, up to the roots of the tree, the nodes are created by hashing together their child nodes (the data blocks themselves for the leaf nodes).

Exploiting this property is possible to generate various types of proof:

- **inclusion proof**: also know as *Merkle proof*, guarantees that the block exists in the chain we are interested in;
- **integrity proof**: ensures that the block has not been modified;
- **consistency proof**: certify that a certain block is consistent in two different time moments.

Since, inclusion and integrity proofs are the ones most closely related to the work which will be analysed in the continuation in the thesis, let us go into how they are created and validated through an example based on the figure 3.3.

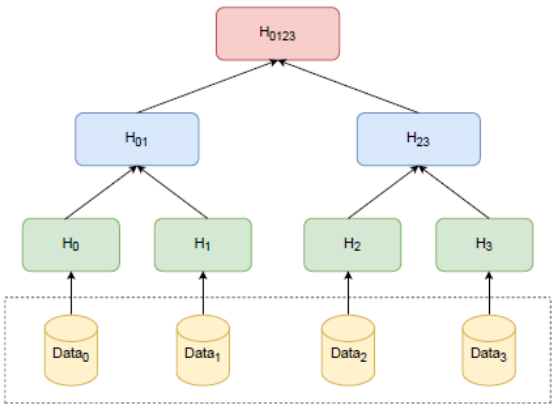


Figure 3.3: Example of Merkle Tree for inclusion and integrity proofs

Essentially these proofs consist of a pair:

$$MerkleProof = (MerkleRoot, [H_i, \cdots, H_y]) \tag{3.1}$$

Where:

- *Merkle Root* is the merkle tree root whose validity you want to prove;
- $[H_i, \cdots, H_y]$ is a set of hashes maintained in some nodes of the *Merkle Tree*, whose validity we want to prove, necessary to validate the proof.

The verification of a proof formed by a pair of this type, consists in reconstructing the *Merkle Tree* which includes the block we want to verify through the hashes provided in the proof. This reconstruction is done starting from the hashes of the leaf nodes up to the root. The root resulting from the tree built by us will then be compared with the *Merkle Root* provided in the proof. If these two roots are equal the verifier can be sure that block is both included in that Merkle tree and integer according to that specific Merkle root. Therefore, this procedure can be used both for inclusion and integrity proofs.

Referring to the 3.3 image, a proof on the data block $Data_1$ could look like this:

$$MerkleProof = (H_{0123}, [H_0, H_{23}])$$

The construction of the *Merkle Tree* for proof verification proceeds according to these steps:

1. compute $H_1$ the hash for data block $Data_1$;
2. compute $H_{01}$ starting from $H_0$ (included in the set of hashes in the proof) and $H_1$ (computed in the previous step);
3. compute $H_{0123}$ starting from $H_{01}$ (computed in the previous step) and $H_{23}$ (included in the set of hashes in the proof);
4. compare $H_{0123}$ (computed in the previous step) with the Merkle root included in the proof ($H_{0123}$).

## 3.3 Cross-chain: Cosmos & Polkadot

As the name suggests, cross-chain solutions aim to bridge multiple blockchains together, enabling communication and data exchange between independent blockchains. Usually, a cross-chain system is made by a number of independent blockchains and bridging technology, such as a *relayer*.

For cross-chain transactions, actually, in the relay model, verification of data regarding cross-chain transactions is abstracted into a relay-level consensus problem. In practice, a blockchain with improved scalability called *relay chain* is developed that takes care of interoperability. The latter will be supported by a number of relay nodes deployed in each blockchain network, which are responsible for monitoring and synchronising the transaction data of that blockchain with the relay chain. The relay chain consensus nodes verify the validity of cross-chain transactions and trigger the execution of corresponding transactions in the destination chain.

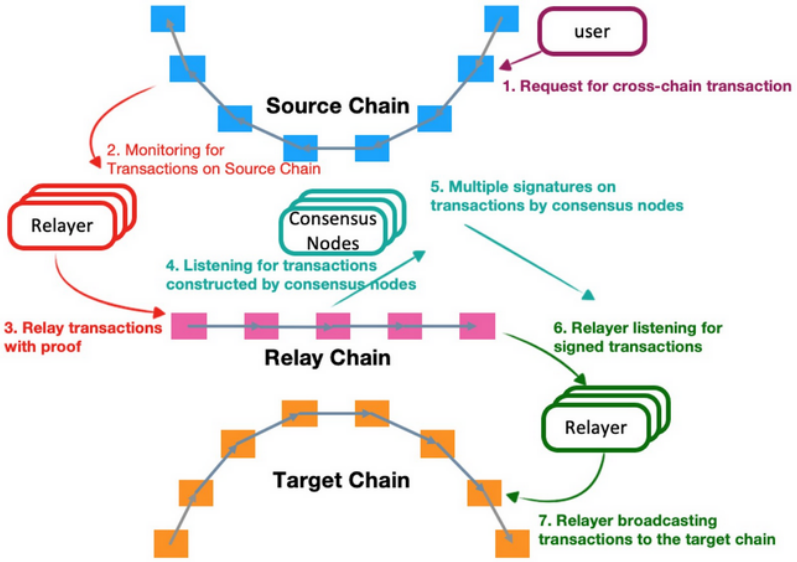


Figure 3.4: Schematic of a relay model from [26]

As can be seen from the figure 3.4, the steps in this relay model are as follows:

1. The user initiates a cross-chain transaction request in the source chain;
2. The relay node in the source network monitors and synchronises the transaction data to the relay chain;
3. The relay chain consensus node verifies the validity of the transaction
4. The consensus node constructs the corresponding transaction
5. A super-majority of consensus nodes signs the transaction, forming a signature set
6. The relay node in the destination network monitors the transactions and signatures
7. The relay node in the destination network broadcast the transaction to the destination chain

### 3.3.1 Cosmos

Cosmos[27] was created with the goal of having an ecosystem of interconnected blockchains. In this ecosystem between different blockchains, assets

such as tokens or funds in general can be exchanged. In Cosmos each chain that is part of the ecosystem is defined as a *zone*. Each of them then will be connected to a central *zone* called *Cosmos hub*.

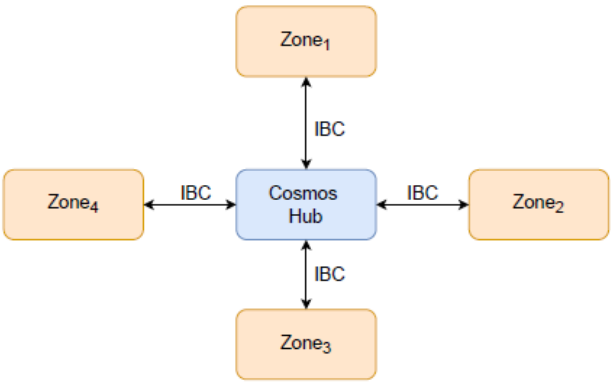


Figure 3.5: Cosmos structure

This hub handles transactions and token exchanges between zones, using a consensus algorithm called Tendermint [28].The connection between zones and *Cosmos hub* is made through a protocol called IBC (*inter-blockchain communication*). The latter consists of a constant packet stream from the zones to the hub to enable it to maintain an updated status of all the zones to which it is connected. Likewise, each zone maintains the state of the hub. As we have mentioned, Tendermint (variant of Byzantine Fault Tolerance consensus algorithm) is used to maintain security between zones, while, Cosmos hub security is maintained through a globally decentralised set of validators, aiming to withstand even the most severe geographical attacks. The entire system can thus be seen as a large blockchain in which each *zone* is an account that can send and receive tokens from another account (*zone*). Each transaction, on the other hand, identifies a data packet exchange which must be paired with a related *Merkle proof*. This proof is used to prove that tokens have been effectively sent by the sender zone. A *zone* cannot spend more than the tokens it has, and every transaction has to be processed from the hub before being committed on another zone. Data transfer works according to these steps:

1. to carry out a transaction, the sending zone sends two types of IBC data packet to the hub:

    - **transaction packet**: contains information about the transaction, such as the sender, the receiver, the amount of involved tokens;

- **proof packet**: contains the proof related to the transaction to be carried out;

2. the hub updates the states;
3. the hub forwards the packets to the receiver zone.
4. the receiving zone can verify the proof on his own using the packet and the sender's headers.

### 3.3.2 Polkadot

Polkadot [29] is an implementation that uses the relay model. Unlike Cosmos, therefore, there is no central hub that connects various zones by managing their transactions, but a central relay chain that bridges heterogeneous blockchains. These independent blockchains located around the relay chain are called *parachains*. Like Cosmos, Polkadot uses an asynchronous consensus algorithm inspired by the classical BFT and Tendermint to enable inter-chain communication and transaction execution.

As you can see from the figure 3.6, Polkadot uses a Proof of Stake [30] consensus mechanism. In the consensus mechanism of the network, there are four roles:

- **validators**: who staking tokens to validate transactions and supervise the stability of the network;
- **nominators**: who select the validators;
- **collators**: manage the nodes that store the entire history of each parachain and compile transaction information into blocks that are sent to the relay chain;
- **fishermans**: are in charge of keeping an eye on the network to eliminate actions that could negatively impact the entire blockchain or its users.

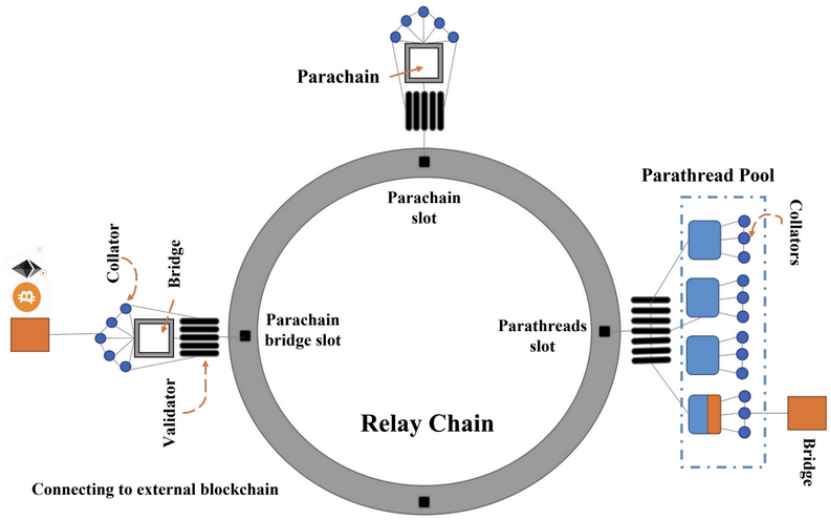


Figure 3.6: Polkadot structure explained in [31]

## 3.4 Side-chains: Polygon

Another solution to improve the scalability of blockchains and decrease their transaction execution load is *Side-Chain*. In this model there is a *Parent Chain* and chains called *Pegged* because they are anchored to the main chain. In case we want to transfer funds (tokens/assets) from the *Parent Chain* to the *Pegged Chain* a so-called *Symmetric 2-way Peg Protocol* shown in the figure 3.7 is used.

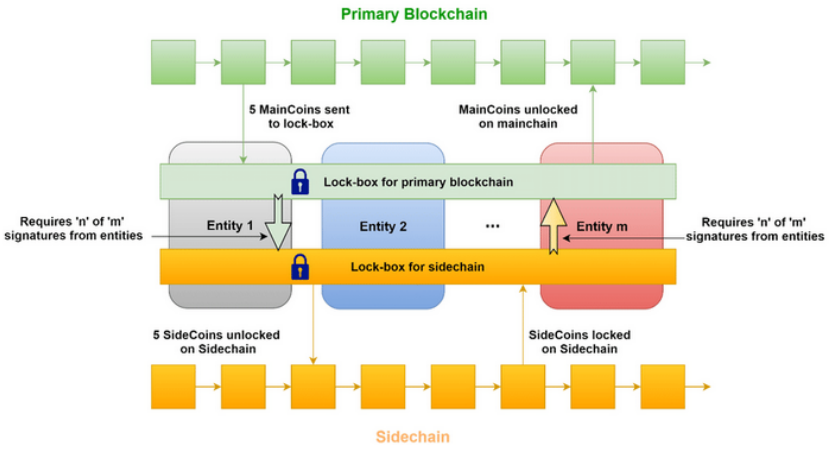


Figure 3.7: Example of 2-way symmetric peg protocol functioning form [32]

This protocol works according to the following steps related to previous

figure:

1. a certain amount of funds are "locked" on the *Primary Chain*, as a security measure to avoid double spending on both parent and pegged chain. They are then inserted into the lock box for primary blockchain;

2. after a confirmation period (in the 3.7 image are required $n$ on $m$ signature from $m$ Entity) the funds are transferred in the lock box for side chain.

3. a proof generation is required before funds are transferred to the *Side (Pegged) Chain*. After another time period, the funds are unlocked and can be freely spent on the *Pegged Chain*.

Being a symmetric protocol the same steps can be taken from the *Side-Chain* to the *Primary (Parent) Chain*.

We bring, below, as an example of implementation, Polygon.

### 3.4.1 Polygon

Polygon[33] is a solution that falls into the side-chain family with full compatibility to the Ethereum Virtual Machine since it is developed on top of Ethereum. The fact that it links well to Ethereum gives the possibility to easily transfer funds from the main Ethereum blockchain to Polygon. Another advantage is that it offers a wide choice of modules and frameworks for the development of DApps based on the Ethereum development stack. Examples of available frameworks include Zero Knowledge engines to identity management modules or Polygon Edge. The latter essentially is a framework for creating side chains pegged to the main Ethereum network. The cross-chain communication protocol used by the side-chains produced with it is interesting and noteworthy. This communication protocol is implemented through a module called *Chainbridge*[34]. The latter is a bridging module (i.e. that acts as a bridge to connect something) based on *relays* that vote to execute requests. The relayers, in turn, use a bridge module made up of three contracts that perform different tasks:

- **bridge contract**: handles requests, votes and the Deposit commands used to transfer assets;

- **handler contract**: sets up the transfer executing setup operations;

- **target contract**: finally executes the transfer on the other side-chain.

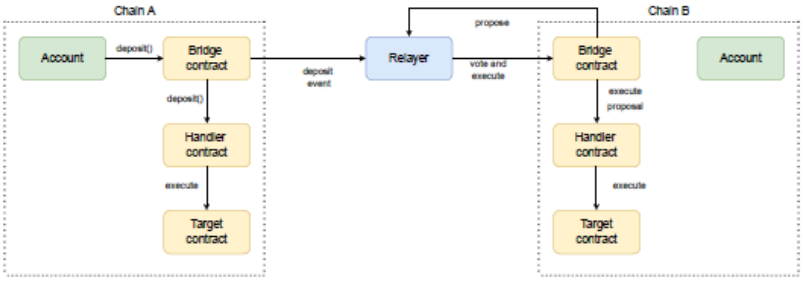


Figure 3.8: Chainbridge exchange protocol overview

## 3.5 Rollups

For last, we mention the *Layer 2* solution called *Rollups*. This solution involves off-chain grouped execution of transactions of a main chain. The results of these executions are then committed to the main chain accompanied by a related proof. The use of rollups enables scaling and consumes less gas to execute a smart contract. In Ethereum, for example, the use of rollups allows off-chain aggregation of transactions inside an Ethereum smart contract. This leads to reduces fees and congestion by increasing the throughput of the blockchain from its current 15 transactions per second to more than 1,000. The proof accompanying the commitment of execution results can be of two categories, that go to define as many types of rollups:

- **Optimistic rollups** whose validity is ensured by **fraud proof**;
- **Zero Knowledge rollups** whose validity is ensured by **zero-knowledge proofs**

### 3.5.1 Optimistic rollups

In the optimistic rollup model, committed off-chain transactions on the main chain are called *trusted*. This means that on the main chain no one is involved in re-executing them to verify their validity. Their validity is verified through this mechanism:

1. the layer 2 application executes off-chain transactions and commits the results on the blockchain network;
2. at this point, off-chain transactions are trusted as valid but not definitely committed on-chain. They remain in a waiting state called *suspension* for a certain amount of time.

3. in this time interval, other users of the blockchain network can prove the **untrustworthiness** of the *suspended* transactions by so-colled *Fraud Proof*. It proves the existence of some error in the execution of the transaction that justifies its *nullification*. In contrast to other systems of proof reviewed in the background chapter, this one is a proof of *invalidity*.

As an example of optimistic rollup we explain Arbitrum[35]. The latter, for *fraud proof* generation uses a protocol called *bisection* 3.9. In this protocol, there are two parties involved:

- **operator**: the user who committed the transaction to disprove;
- **challenger**: the user who wants to generate the *fraud proof*.

The transaction is modelled as a change of state, starting with some preconditions and ending in some post conditions through the execution of a set of N instructions as from the image 3.9. At each step of the protocol:

1. **operator**: must find two valid subsets of size N/2 that compose an intermediate state. The latter will serve as the post condition of the first subset and the precondition of the second;
2. **challenger**: must indicate which of the two subsets found by the operator contains the error in the transaction. The chosen subset is used as a precondition for the next iteration.

This procedure is iterated until one of these three options takes place :

- the **operator** cannot find a valid split for a certain subset. This leads them to lose the challenge and thus the transaction is disproved.
- the **challenger** fails to find any error in a pair of valid subsets. So it loses the challenge and the transaction is committed;
- an iteration results in two valid subsets that contain only one instruction each. In this case, the **challenger** chooses the instruction identified as an error, and this single instruction will be executed on the main blockchain to determine the winner of the challenge.

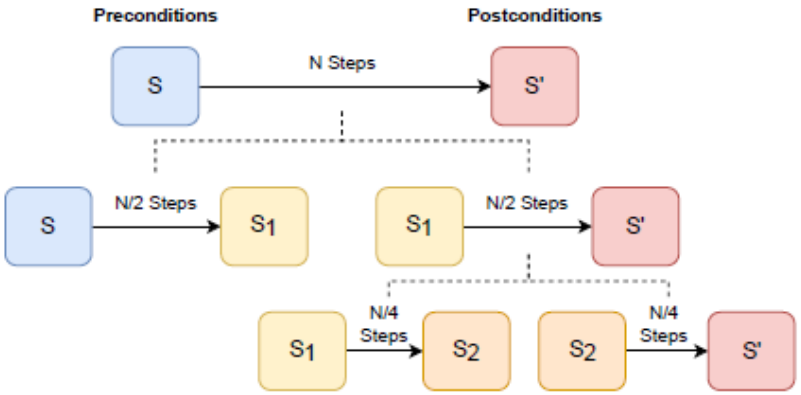


Figure 3.9: Arbitrum bisection protocol example

To make the protocol secure against fraudulent fraud challenges, it is required to the challenger to lock a certain amount of stack. The challenger will lose this locked stack if he is unable to generate a legit *fraud proof*.

### 3.5.2 Zero Knowledge rollups

Zero Knowledge rollups[36] were designed as an alternative to *Optimistic* rollups to scale even more Ethereum. This is because, even though there will be more computation on the chain with this type of rollup, it will eliminate the waiting time due to the *suspension* period of transactions in the *Optimistic* rollup model. Another difference that characterises it, compared to *Optimistic* rollups, is the use of a "traditional" proof model, this means that, every transaction executed outside the main chain, will be committed to it together with a proof of validity attesting its integrity.

This proof of validity in the *Zero Knowledge* model are called *Zero Knowledge Proof* (ZKP). This knowledge proof ensures the correctness of a certain logical predicate related to the information to be proved but without revealing it in concrete. An example of application may be the following. Starting from a logical predicate such as, "My balance is greater than 100.000?"

ZKP certifies the rightness or wrongness of this condition (through, for example a Boolean flag) but without revealing the actual amount of the balance itself.

The most widely used implementation of ZKPs at the moment is the zkSNARK (*Scalable Non-interactive ARgument of Knowledge*) proof protocol. As an implementation of ZK rollups that uses this protocol, we mention

*ZoKrates* [37]. This framework, automatically generates smart contracts for verifying zkSNARK proofs directly on-chain.

Alternatively, it allows the generation of proofs for off-chain transactions by going to eliminate the "suspension" time required by *Optimistic* rollups but requiring some on-chain computation. Transactions are then actually committed based on the result of the proof.

ZoKrates uses a high-level programming language to define some arithmetic and logical predicates that can be used in the proofs, using public or private parameters as inputs as shown in the figure 3.10. We want create a ZK proof for the predicate: "is my balance greater than 100,000?"

The private parameters are passed by the *prover* during proof generation, while the public parameters are used to verify the proof. In the above figure 3.10, the actual user account balance (150.000) will be a private parameter, while the required threshold (100.000) will be a public parameter. The proof generated by *ZoKrates* guarantees the answer "YES" for the predicate.

ZoKrates, before generating a proof needs a setup phase which consists of sharing a secret key between the *prover* and the *verifier*. This will be used, by the *prover* to generate the proof and by the *verifier* to verify it. Proof generation can be performed by any DApp that integrates *ZoKrates* into its logic, while verification can be performed by implementing an on-chain smart contract.

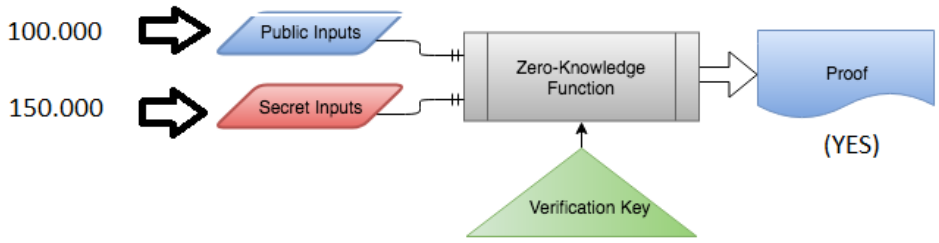


Figure 3.10: Example of ZK proof generation with *ZoKrates*

# Chapter 4

# CARONTE: a protocol for authenticated cross chain data transfer

After an in-depth look at the world of blockchain (Ethereum in particular) and some successful implementations related to Layer 2 solutions, we move on to the actual project addressed in this thesis work.

The project is inspired by authenticated queries. These queries are used in asymmetric protocols that exploit verification objects (*merkle proof*) present on the chain that answers the query (i.e. *sender* of information). Then, these verification objects are used to check the query results and thus to attest their correctness, on the chain requesting the queries (i.e. *receiver* of information).

Our proposal is to adopt the same model to provide secure cross-chain communication of information via a Layer 2 solution. Communication in this scenario means exchange of information. Our intention is to allow entities (e.g. smart contracts) in the receiver chain to read and act on information stored within the source chain. We want to allow this, either between two side-chains or between a side-chain and a main-chain.

At this point, it is important to specify that the protocol (as for authenticated queries) will be asymmetric. Asymmetric means that information flows in only one direction, namely from the *source* of the information to the *destination*. We translate this property directly from the world of authenticated queries between light and full node in which there is a clear separation between the initiator of the query (light node) and the responding entity (full node). This is because, within the protocol, the two types of nodes have a different degree of risk to bear. The light node is the entity interested in having reliable information so it must use verification objects to check its

validity. Indeed, it relies on a potentially unreliable node to retrieve information from a chain over which it has no visibility. The full node, on the other hand, does not bear any risk from participating in the protocol. This is why the two parts of the protocol behave differently and cannot request and send information symmetrically.

The fact that the source chain of information is completely inaccessible from destination user gives to the model the possibility of running between any combinations of public and private chains. Since there is no risk of access from third party users, a source chain may actually be private.

However, even if the basic protocol allows a one-sided flow of information, it does not mean that two instances of the same cannot be used. There is always the possibility of using another instance of the protocol with switched recipient and source roles to allow information to flow the other way as well.

Of course, to ensure all the properties of a regular blockchain, the communication mechanism provided will have to be trustworthy. Trustworthy, in this case, will mean that, in the exchange of information no reliability is assumed by the respective chains (mainly from the *source* chain). This is because it is the exchange protocol that will ensure that the data exchanged between one chain and the other is reliable.

Since we want to allow the information produced in the source to be consumed in the destination by ensuring its reliability, it will also be useful to save a concise representation of the source in the destination in order to be able to verify the correctness of the messages. This is in line with the asymmetric nature of the protocol, in that the protocol only affects the destination, leaving the source completely intact.

We have chosen to call our cross-chain information exchange protocol *CARONTE* (Cross-chAin infoRmatiON auThenticated Exchange). The name seems to us to fit particularly well since, like the popular ferryman from the underworld who required an obolus to escort the dead from one side of the river *Acheronte* to the other, our protocol will have to ensure integrity and reliability in cross-chain data transfer.

## 4.1 Problem Description

The project deals with the issue of trust in exchange of information between 2 chains, that we can call *C1* and *C2*. These, can be either Ethereum's main chain or its sub-chains. Specifically, we want to enable *C1* to retrieve variables or information contained in a contract saved on *C2*, without needing to download all the blocks of *C2* to make sure that the information passed to it is correct.

For this purpose, we developed a *Smart Contract* deployed on *C1* and that would act as a *"Bridge"* from *C1* to *C2*. This *Smart Contract* will be responsible for carrying out all the necessary checks to insure validity of the data exchanged between the two chains. We will refer to this bridge contract as *bridge smart contract C1-C2* since, although physically present on *C1*, it is a bridge which transfers validated C2 information, from C2 to C1.

## 4.2 Structure of CARONTE

The various steps to be taken to exchange information in trust mode are depicted in the image below.

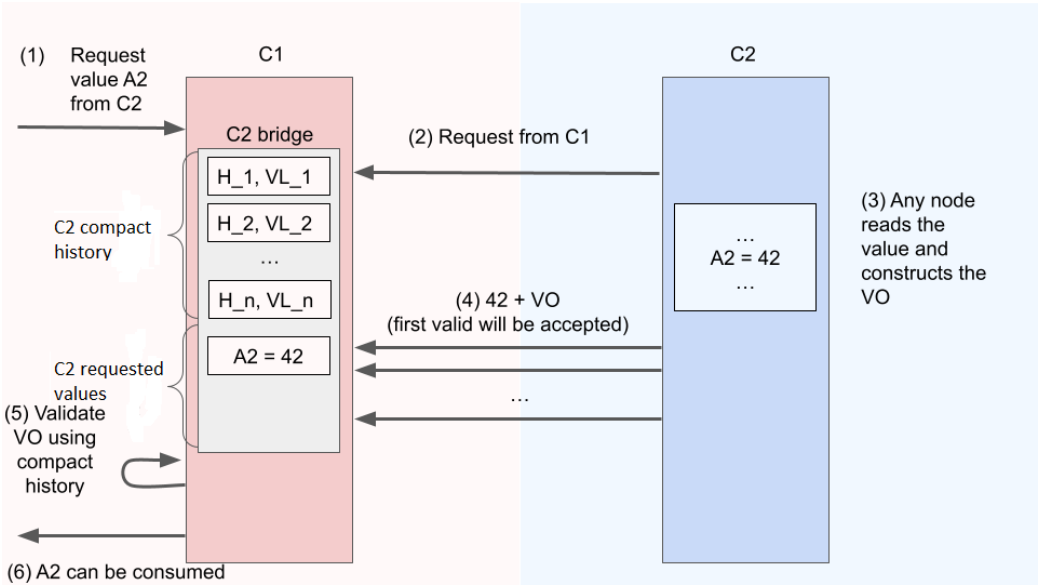


Figure 4.1: CARONTE protocol

In the Figure 4.1, when we talk about *C2 Compact History*, we are referring to the fact that, the *bridge smart contract C1-C2* maintains in its internal state a kind of summary image of the *C2* blocks. This is done in order to ensure the passage of information in a reliable manner. The information that makes up these *C2* blocks images, before being added to the internal state, will be validated by the *bridge smart contract C1-C2* to ensure its trustworthiness.

The steps of cross chain exchange protocol CARONTE we figured out are:

1. someone on *C1* requests an information (variable) contained in *C2*;

2. *C2* receives the request;

3. any node on *C2* takes over the request, reads the requested information, and produces a verification proof for that information. This proof can be used by *C1* to ascertain the validity of the information;

4. *C2* user sends the information with attached proof of its validity to *bridge smart contract C1-C2*;

5. the *bridge smart contract C1-C2* verifies the proof attached to the information received through C2's compact history that the contract maintains.

6. if the proof passes the validation, then, the related information can be consumed on the *C1* chain.

In addition to the *Bridge* contract, which implements the CARONTE protocol, "Entities" located on *C1* and *C2* were also created. Their purpose is to mimic the behaviour of users, on their respective chains, using the *Bridge* contract and thus our proposed exchange protocol. This then leads them to call the methods of the *Bridge* contract to test the proper functioning of the information exchange protocol exposed above. They are three:

- **Requester**: Artificial user of *C1* that mimics the behaviour of a user who wants to retrieve information about *C2*. In order to do this, it executes the *Bridge* contract. Specifically, it executes a message call type transaction that invokes the execution of the `request` method of the *Bridge* contract;

- **BlockSaver**: As mentioned earlier, the *Bridge* contract to enable secure exchange of information will need the compact history of *C2*. This will consist of a summary of the blocks headers of the *C2* chain. To acquire it, the contract offers a method called `saveBlock`. This is where our artificial user BlockSaver comes in. It impersonates those users who, for a reward (ETH), take care of executing the contract by calling the `saveBlock` method. This is for the purpose of keeping the compact history of *C2* updated in the bridge contract. This method will create a win-win situation both for the *Bridge* contract, whose compact history of *C2* will be kept up-to-date, and for the users of *C2* who will receive a reward for their service;

- **Listner**: Last but not least, the *Listner* mimics the *C2* users who will be responsible for validating the information passed through the *Bridge*

to the *C1* chain. Specifically, it, whenever the *Bridge* contract notifies it of a request that has arrived from *C1*, takes care of:

1. searching for the requested information;
2. execute the *Bridge* contract by calling the `verify` method, which will take care of validating within the contract the data passed by the *Listner*.

Of course, the *Listner*, according to the structure envisioned for this *Layer 2* solution, will also receive a reward in return whenever it takes care of performing these tasks.

We still don't go into the depth of how the validity of information is maintained. Let's just say that each information is followed by a *proof* of its validity. It may happen that a user on *C2* who acts as a *BlockSaver* or a *Listner* tries to forge the data passed to the contract. In this case the *Bridge* contract, thanks to its control mechanisms, will not accept the data passed by malicious users and consequently they will not even receive a reward for their work. Malicious behaviour is therefore discouraged as it will lead to wasted work.

Below you will find sequence diagram showing the interaction between these "Entities" according to the exchange protocol stated above.

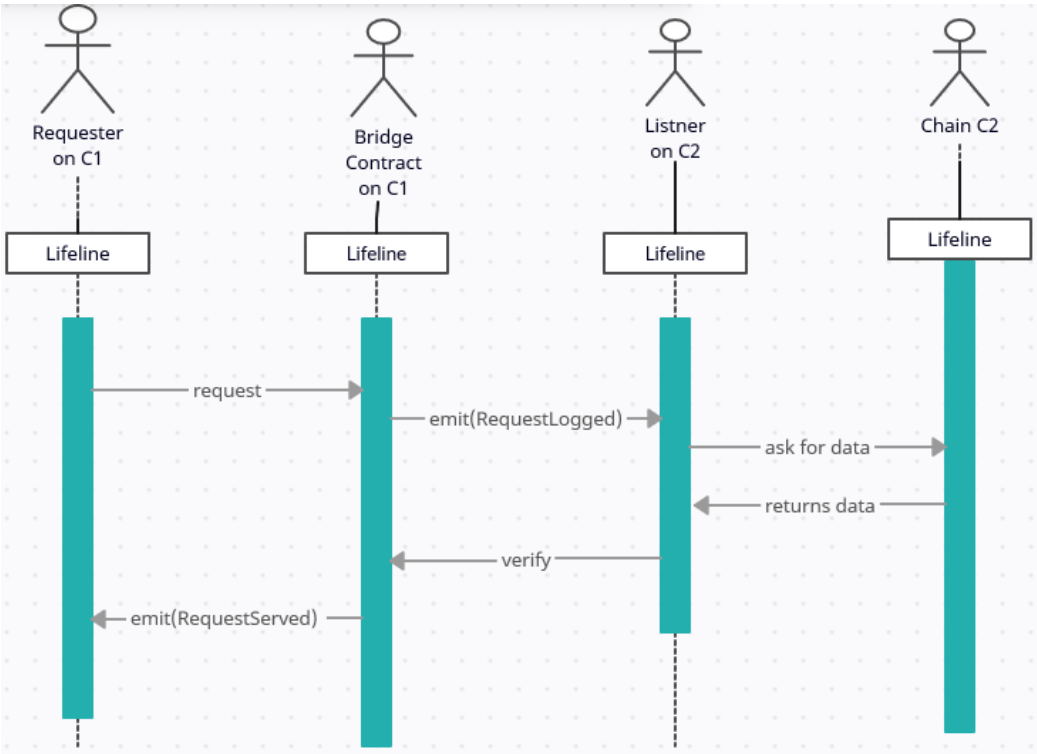


Figure 4.2: Sequence diagram of exchange protocol

From the figure 4.2 we can see these steps in sequence:

1. The actor *Requester* executes the `request` function of the *Bridge* contract to request information on *C2* chain. It then waits for a `RequestServed` event to get the requested data back;

2. The *Bridge* contract during the execution of the request function emits an event called `RequestLogged`;

3. This was precisely the event on which the *Listner* was waiting which, upon receiving it, contacts the *C2* chain to retrieve the request data;

4. after finding the data on chain *C2*, the *Listner* executes the verify method of the contract;

5. if the information with the relative proof passes the verification checks carried out in the `verify` function of the contract *Bridge*, then a `RequestServed` event will be emitted (the one on which the *Requester* was waiting).

Finally we show another sequence diagram showing the update of the compact history of *C2* on the contract *Bridge*.

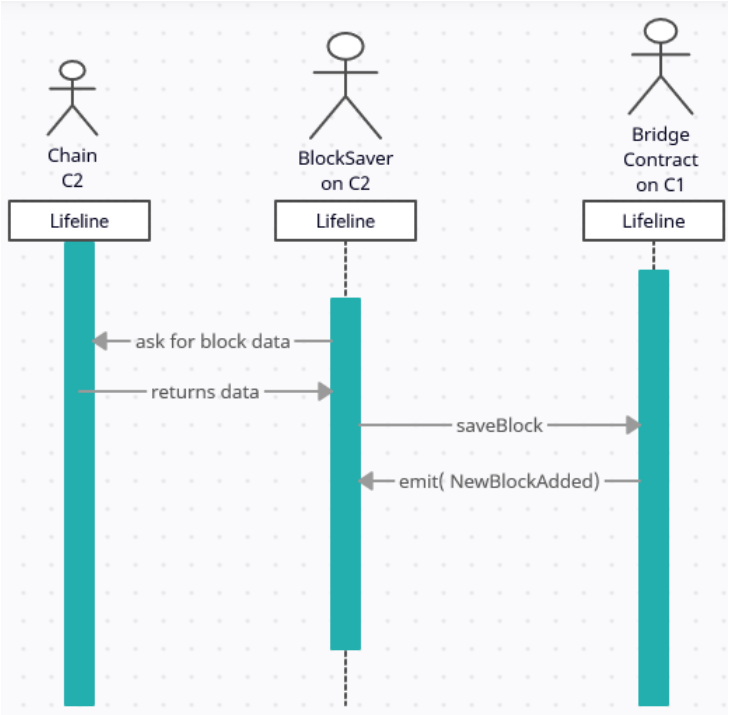


Figure 4.3: Sequence diagram regarding the update of the compact history of *C2* in the *Bridge* contract.

First, let's say that of course the *BlockSaver* may not always be the same user on chain *C2*. Furthermore, this sequence of steps will be repeated every time a new block is added to *C2*. Also in this case we explain the 4.3 image using a sequence of steps. These take place after a new block has been added to *C2* so its compact history on the *Bridge* contract will need to be updated.

1. The *BlockSaver* contacts the *C2* chain to get the interesting data (header) about its newly added block. Among them there is also the `hash` field that is the result of the formula:

$$block.hash = keccak256(RLP(headerFields)) \tag{4.1}$$

2. After receiving this data from *C2* chain, the *BlockSaver* executes the `saveBlock` function of the *Bridge* contract. It then waits for a `NewBlockAdded` event to ensure that the compact history has been correctly updated in the *Bridge*;

3. while executing the `saveBlock` function, the contract performs all the necessary checks to ensure that the information received is correct. The integrity of the headers to be saved in C2's compact history in *Bridge* contract is carried out according to these steps:

   (a) we have the header fields of a block available in plain text ($[f_0, \ldots, f_n]$);

   (b) according to the encoding mode determined by RLP in the yellow paper [6], for each of these fields RLP encoding is performed producing: $headerFilds = [RLP(f_0), \ldots, RLP(f_n)]$;

   (c) then $RLP(headerFilds)$ is produced;

   (d) we finish calculating the hash according to the header fields passed to us in the clear, applying the *keccak256*[12] encoding function to $RLP(headerFilds) : ourHash = keccak256(RLP(headerFilds))$;

   (e) finally we compare $ourHash$ with the hash passed by the blockSaver when calling the `saveBlock` function. If these match, the header passed to us is valid. It will not be valid in the opposite case.

4. If the header passed is valid, the contract emits a `NewBlockAdded` event (the one the *BlockSaver* was waiting on).

# Chapter 5

# CARONTE: implementation

We now turn to examine the concrete development work. This is aimed first of all at the development of the *Bridge* contract that concretely implements the exchange protocol examined in the previous chapter. This contract, was developed using the Solidity language [15] in several versions, which we will explore in more detail later in the chapter. It will be referred from now on as `Bridge.sol`. We then moved on to the implementation through `python` [38] of the three entities: *Requester*, *Listner* and *BlockSaver*. Their implementation has been used to test the proper functioning of the `Bridge.sol` smart contract. The complete implementation code for the bridging module, developed for this thesis, can be found at [1]. `Bridge.sol`, uses some libraries, again developed via smart contract, which we will examine.

## 5.1 Utility libraries

### 5.1.1 RLPReader

This contract library, available at [2] provides an interface to first take RLP encoded bytes and convert them into an internal data structure, `RLPItem` through the function, `toRlpItem(bytes)`. This data structure can then be destructured into the desired data types as `List`, `Address`, `Bytes`, `Uint`, ecc.

This library will be used in `Bridge.sol` to manipulate data from the Ethereum chain that makes extensive use of RLP encoding [6]. As we saw in the chapter 2, RLP encoding is used, in Ethereum, to represent the structured format of trie nodes as arrays of bytes making these nodes suitable for hashing and persisting.

[1] Bridge.sol.https://github.com/gimmydr97/MasterThesis
[2] RLPReader.sol.https://github.com/hamdiallam/Solidity-RLP/blob/master/contracts/RLPReader.sol

### 5.1.2 MerklePatriciaProofVerifier

As specified in the previous chapter, each piece of data passed to the `verify` function of the `Bridge.sol` contract is accompanied by a proof to enable its verification. So, this library offers the facilities to verify the proofs coming from the C2 chain and thus ensure that the information exchanged with C1 are reliable. `MerklePatriciaProofVerifier.sol`, available at [3] provides the `extractProofValue` method that deals with the validation of proofs coming from C2. This function will be covered later in this chapter concerning the `verify` method we implemented.

### 5.1.3 StateProofVerifier

We used the methods and structures offered by this library as the basis for implementing the verification part of our `Bridge.sol` contract. It too can be found at [4] Among the functions it offered there is `parseBlockHeader`, which takes as input a block header in the RLP encoding form. Via the `RLPReader` library it extracts the block header fields and saves the most important ones in a `StateProofVerifier.BlockHeader` structure, which then returns. The `BlockHeader` structure as well as all the other structures and functions offered by this library will be covered in detail in later sections discussing the implementation of `Bridge.sol`.

## 5.2 Bridge Contract

We now turn to the implementation of the `Bridge.sol` smart contract (related to appendix A). This contract, to implement the cross chain exchange protocol discussed in the previous chapter provides, three methods : `request`, `saveBlock` and `verify`. These three are used to fulfil the three main tasks that the `Bridge.sol` contract must perform. These are:

- save all the requests received from *C1* users, for trust information retrieval on *C2*, in an internal state so they can be processed asynchronously;
- save the compact history of C2. This function is periodically called to update an internal structure that maintains all the headers of the new

[3] MerklePatriciaProofVerifier.sol.https://github.com/lidofinance/curve-merkle-oracle/blob/main/contracts/MerklePatriciaProofVerifier.sol
[4] StateProofVerifier.sol.https://github.com/lidofinance/curve-merkle-oracle/blob/main/contracts/StateProofVerifier.sol

mined blocks on C2. This structure, then, will be used as a lightweight depiction of C2 chain. This will be useful to verify the proofs attached to information arrived from C2;

- verify the autenticity of the information received as a response from C2 by actually checking the verify proof related to it.

### 5.2.1 Data Structure

First, it is important to say that three different solutions were proposed in the `Bridge.sol` contract implementation process, which will be compared later in chapter 6. These three implementations differ depending on:

- variables that we can call "auxiliary," useful for maintaining the internal state of the requests and compact history of the C2 chain;
- what data structure was used to maintain the compact history of the C2 chain.

In accordance with this consideration we are going to list the three implementations with corresponding internal states.

#### Bridge basic

This is the first implementation developed, with no optimization. The data structures used are those that we had initially considered necessary for the contract to function properly without, however, worrying about its performance. Its internal state is formed as follows:

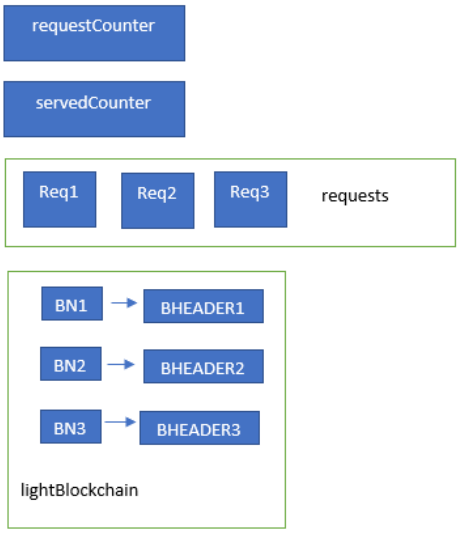


Figure 5.1: Internal state of the *basic* `Bridge.sol` smart contract

The internal state therefore consists of `requestCounter` and `servedCounter` which are respectively a counter for requests received and a counter for requests already served. Then, there is an array in which all received requests are stored called `requests`. Finally, there is a key-value map that saves *C2*'s compact history in the internal state of the contract. It uses the block numbers of the blocks as a keys. Map values are instead the encoding of block headers.

### Bridge with Sliding Window

This implementation attempts to decrease the space used by the map that maintains C2's compact history. It does this by using a sliding window algorithm [5].

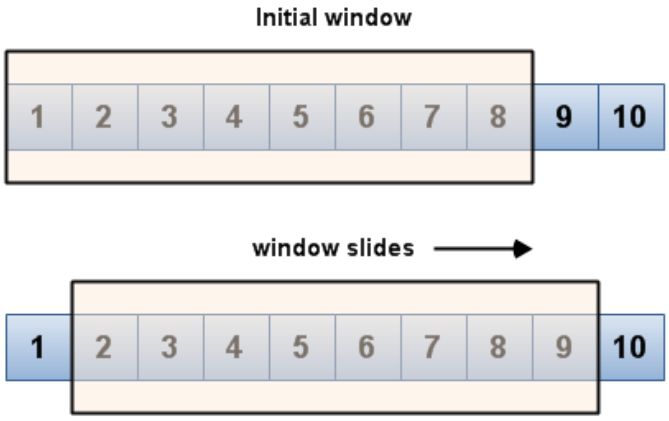


Figure 5.2: Example of functioning of sliding window

Essentially whenever a block header of *C2* is too outdated and is still maintained in the compact history this is removed from the map to save storage space. Below we explain when a block is considered outdated. We start with the idea that a user requesting information wants to find it in its most up-to-date state. It is also useful to know that a block, on average, is added to an Ethereum-based blockchain every 15 seconds. Starting from these two information based on the amount of requests arriving at the contract bridge every second, we should calculate an amount of blocks to remain in C2's compact history to ensure that even if a request is served late, the block on which the request was made is still in the compact history. This quantity of blocks will constitute the size of the window. This size is parameterised by being decided during the deployment of the `Bridge.sol` contract by passing a parameter `windSize` to the contract constructor.

The **sliding window** will allow us to have only the info related to the most recent `windSize` block of *C2* in the C2's compact history of `Bridge.sol`.

For the proper functioning of the sliding window algorithm, additional variables and structures were added to the internal state. These are:

- `windSize` which constitutes the size of the sliding window decided upon during deployment;
- `blockCounter` which counts the total number of blocks saved in the contract;
- `window` which constitutes the array that holds the block numbers of the blocks that are present in the sliding window at a certain time.

#### Bridge with array as compact history

In the last proposed solution, we exploit a structure not too expensive in terms of memory, not necessarily dynamic but with the right size to allow processing of all the `verify` that arrived in a certain amount of time. In this regard, the map structure is too powerful and unnecessarily heavy even with the contribution of the sliding window policy. We have, therefore, decided to replace it with a less expensive *array of fixed size*, to maintain the compact history of C2. This size is decided during deployment of the contract according to the `arrSize` parameter of the constructor. The use of a normal array, however, makes the search for the different references to C2 blocks within it, more complicated to develop. In this case we will need other supporting variables to facilitate the use of the array instead of the map. With respect to the internal state of the basic solution, we therefore have that :

- `lightBlockchain` from a map becomes an array of fixed size;
- we add `arrSize` which is the size of the array decided at deployment time;
- we add `bockCounter` which holds the total number of blocks stored in the contract;
- we add `lastKey` which holds the block number of the last block saved in the compact history of C2.

### 5.2.2 Function implementation : request

We now move on to define the various methods that `Bridge.sol` implements, starting with `request`. This is the function invoked to retrieve information about C2. It takes charge of the request and adds it to the internal state of the smart contract. This below is the pseudocode which summarises the main actions it performs.

```
1    function request(parametes){
2        //create a request instance
3        request = requestInstance(parameters);
4        //add it to the requests array
5        requests.add(request);
6        //update the counter of requests received
7        requestCounter++;
8        emit RequestLoggedEvent;
9    }
```

As seen above, for received requests, the internal state consists of an array named `requests`. The latter is an array of `Request` structures addressed by

a request counter `requestCounter` that is updated for each received request. The `Request` structures maintain:

- `account`: address of the *C2* contract we want to read data from;
- `key`: numeric identifier of the *C2* variable we want to read;
- `blockId`: identifier of the *C2* block we want to read data from;
- `date`: timestamp of the request;
- `served`: a Boolean value which is equal to true if the request has been served, false otherwise;
- `response`: the value associated with the variable that the user on *C1* is searching for.

Of course not being served yet, the request, is saved with the field `served = false` and `response = 0` at `requestCounter` index. Actually, the function takes care of issuing an event `RequestLogged(requestId, account, key, blockId)`, update the `requestCounter` and return to the caller the `requestId`. The latter is the index of the array to which the request will be retrieved.

### 5.2.3 Function implementation: saveBlock

This function is responsible for saving C2 block headers in the internal state of the `Bridge.sol` contract, which, in this case consists of a data structure named `lightBlockchain`. The latter, takes as its key an integer consisting of the numeric block identifier and as its value a structure of type `StateProofVerifier.BlockHeader` containing:

- `hash`: the hash of the block calculated via Keccak256 [12] of the RLP [6] encode of the header,
- `stateRootHash`: the hash representing the root of the block,
- `number`: the numeric identifier of the block in the blockchain,
- `timestamp`: is the epoch Unix time of the time of block initialisation.

The function is designed to be called periodically to update the internal state whenever a new block on C2 is added. Its operation can be summarised by this pseudocode.

```
1     function saveBlock(parametes){
2         // verifies the correctness of the head
3         newBlockHeader = verifyBlockHeader(pameters);
4         // only for sliding window and array solutions
5         if(blockCounter > size){
6             toDelate = blockCounter%size;
7             lightBlockchain[toDelate] = newBlockHeader;
8              blockCounter ++;
9         }
10        //only for basic solution
11        ligthBlockchain[newBlockHeader.blockNumber] =
              newBlockHeader;
12
13        emit NewBlockAddedEvent;
14    }
```

In actual implementation, it takes as a parameter a data structure `Bridge.BlockHeader` containing:

- the hash of the block to be added to the internal state,
- the RLP encoding of its entire header.

The implementation of this method, varies depending on the solution (basic, sliding window, array). Generally it performs these steps to store a new `StateProofVerifier.BlockHeader` to `lightBlockchain` structure.

1. Create the data type `StateProofVerifier.BlockHeader` potentially to be added into the structure. It does this by calling up the auxiliary function `verifyBlockHeader` to which it passes the structure `Bridge.BlockHeader` passed as a parameter to the function. This auxiliary function in turn deals with two steps:
   (a) create the `StateProofVerifier.BlockHeader` structure via the `parseBlockHeader` method offered by the `StateProofVerifier` library. This structure will go to represent the block header potentially to be added to the structure;
   (b) perform validity check on this `StateProofVerifier.BlockHeader`. In particular, it compares the hash of the newly created structure with the one passed by the *BlockSaver*. If the two are equal then it mean that the block has not been tampered.
2. after creating the element potentially to be added to `lightBlockchain` called `bHeader`, the implementation of its inclusion varies depending on the solution.

- **basic**: in this implementation the element `StateProofVerifier.BlockHeader` is simply added to the structure `lightBlockchain` to the key `bHeader.number` (in this case the compact history of C2 is a map);
- **sliding window**: in this case in addition to adding the `bHeader` structure to the `lightBlockchain`, following the sliding window algorithm [5], we have to worry about removing another one. This of course, only happens if the items in the `lightBlockchain` exceed `windSize`. In this case, a support variable `toDelate` is used to retrieve the obsolete block number (to be deleted) from the `window` array. Instead, the block number of the `bHeader` to be added to the `lightBlockchain` is inserted. Through `toDelate` deletes the header of the obsolete block in the `lightBlockchain` structure. Finally, `bHeader` is added to `lightBlockchain` at `bHeader.number` key;
- **array**: As in the previous case, also in the implementation with array we have a maximum number of items to maintain in `lightBlockchain` identified this time by `arrSize`. Related to this, to keep maximum `arrSize` items in `lightBlockchain`, we use the `blockCounter` variable module `arrSize` to cyclically insert blocks always in the same positions of the array.

  $$position = counterBlock\%arrSize \in [0, arrSize) \tag{5.1}$$

  Given this, whenever `blockCounter` exceeds `arrSize` the most obsolete block will be deleted from `lightBlockchain` and `bHeader` will be inserted in its place. After this, `blockCounter` is incremented and `bHeader.number` is saved in `lastKey`. It is important to keep `lastKey` updated since it will be useful to retrieve `lightBlockchain` elements based on their bock number.

3. Once the item has been added correctly to `ligthBlockchain`, an event `NewBlockAdded(hash, stateRootHash, number, timestamp)` is raised and numeric identifier of the added block is returned.

### 5.2.4 Function implementation: verify

This function is the one that executes the concrete verification on the proof that arrives as an attachment with the requested information from C2 that is the internal state variable of some contract located on C2. Its functioning can be summarised by this pseudocode.

```
1     function verify(proof, blockId, requestId){
2
3         if(blockIsPresent(blockId) == true){ //the request-
              related block is present in the compact history of
               c2
4             // verification on the existence of the account
                  linked to the requested information
5             account = verifyAccountProof(proof);
6             // verification on the validity of the requested
                  information
7             requestedInfo = verifiyStorageProof(proof);
8             //update the state of the request linked to this
                  verify
9             requests[requestId] = requestedInfo;
10            emit RequestServedEvent;
11        }
12
13
14    }
```

It takes as parameters a `requestId`, a `blockId` and a `StateProof` data structure containing within it:

- `account`: address of the C2 contract containing the variable whose value is requested;
- `accountProof`: proof that certifies the existence of the contract;
- `storageRoot`: root of the storage tree;
- `key`: location of the requested variable;
- `value`: value of the requested variable;
- `storageProof`: proof that certifies the existence of the variable inside the contract.

This method acts according to these steps:

1. First of all, the function, requests that the `blockId` passed as function parameter identifies a block within the `lightBlockchain` structure. This will be essential in order to perform an effective check on the proof that arrived as an attachment with the requested information (`StateProof`). This is done by calling the `blockIsPresent` control function. The latter, through the block identifier (`blockId`) required as parameter, tries to find the requested block within `lightBlockchain`. If this check is not passed, a `BlockNotFound(blockId)` event is raised

and the request is saved as rejected. The whole state proof verification must fail in this case because to perform it safely, we mandatorily need a `stateRootHash` field of a block that is present in the internal state of the contract;

2. Instead, if this first check is passed, we move on to the actual verification of the proof attached to the requested information. This consists of 2 steps :

   - **the Proof account verification**: this first verification is necessary to validate the real existence of the contract from which we request the information. It is performed through the function `verifyAccountProof` that, after parsing the account proof in a list of `RLPReader.RLPItem`, calls the `extractAccountFromProof` function of the external library `StateProofVerifier`. The latter takes as its parameters:
     - the encoding via `Keccak256` of the `account` of the contract from which we requested the information;
     - the field `StateRootHash` related to the `lightBlockchain`'s block at the `blockId` index;
     - the field `accountProof` passed as a parameter in the StateProof structure, which is the actual proof to be verified. It's basically an array of rlp-serialized Merkle Tree nodes, starting with the StateRootHash node.

     This function, in turn, calls `extractProofValue` from the external library `MerklePatriciaProofVerifier`, which will go on to perform the actual verification on the `accountProof` via a *Merkle proof*. The verification consists of building a **Merkle Patricia Trie** data structure via the hashes contained in the `accountProof`. The `StateRootHash` in our possession will be the starting point from which to traverse the proof Merkle trie, while, `account` will be the path to follow to traverse the tree. If by the completion of this procedure a leaf of the trie is reached, the proof will be valid and we may extract the data for the account we have just verified.
   - **the Storage Proof verification**: this second verification will validate the existence of the requested variable in the previously verified contract. It is performed through the function `verifyStorageProof` that, after parsing the storage proof in a list of `RLPReader.RLPItem`, calls the `extractSlotValueFromProof` function of the external library `StateProofVerifier`. The latter takes as its parameters:

- the encoding via `Keccak256` of the `key` field of the StateProof passed as a parameter;
- the `storageRoot` field of the `StateProof` passed as a parameter. All storage will deliver a Merkle Proof starting with this rootHash.
- the `storageProof` field passed as a parameter in the `StateProof` structure, which is the actual proof concerning the variable request. Actually it is array of rlp-serialized MerkleTree-Nodes, starting with the `StorageRootHash` node.

As above, this function also calls the `extractProofValue` function of the external `MerklePatriciaProofVerifier` library, which will perform the verification, this time, however, on the `storageProof`. after building the*Merkle proof* and making sure that the `storageRoot` coincides with the root of the verification tree (Merkle proof), the `key` is used as path to follow to traverse the tree. The operation of the verification is the same as the one described above, and this time it will allow us to extract the value we had requested from sub-chain C2 related to location `key`.

If either of these two tests fails, the verification will be invalid and thus the value requested from chain C2 will not be reliable.

3. Instead, in the case where both verifications are passed, we will update the status of the request related to the `requestId` passed as a parameter to the function. This will be done by setting the `served` field of the request to `true` and updating the `response` field with the value extracted above. The function, then raises an event `RequestServed(requestId, account, key, blockId, reply)` and finally returns `true`.

### 5.2.5 The getter functions

There are several getter functions in the contract to get information about the internal state maintained. They are:

- `getTotal()`: returns the number of total requests received by the contract;
- `getServed()`: returns the number of requests already served by the contract;
- `getPending()`: returns the number of pending requests. The ones not yet served by the contract;

- `getRequest(uint id)`: returns the `Request` structure related to the request with `requestId = id`;
- `getBlock(uint blockId)`: returns the structure `StateProofVerifier.BlockHeader` related to the block number `blockId`.

## 5.3 Requester

To test the `Bridge.sol` contract, we implemented via Python an artificial user of C1 chain called `Requester.py` (related to appendix B). Its main is send requests to `Bridge.sol` contract via the `request` function.

This artificial user, first, establishes a connection via the `web3` library[39] to C1. We will use, that one, to deploy and fetch a reference to `Bridge.sol` on C1. The deployment of the bridge contract on C1 can also be effected by a third party. In this case the requester only needs to know the address where to find the contract.

Regardless of which alternative was used, our "user" will end up with a reference to the `Bridge.sol` contract that we can use to make calls to the functions it provides. The code then proceeds with an infinite loop witch asks for the data about the information on *C2* chain to retrieve. These are:

- the `address` of the contract from which the user would like to read the variable;
- the `key`: when the deployment of a contract is done, its bytecode is sent via a transaction to the Ethereum blockchain where it will be saved perpetually. The variables that make up a contract's state are saved in its Storage section. The smart contract *Storage Trie* is a key-value mapping, where the key corresponds to a slot number in the storage and the value is the actual value stored in this slot. Solidity store variables in this slot depending on the type of the variable. The storage slots are assigned continuously, starting with slot 0, in the order in which the state variables are defined. Thus they can be referenced by increasing keys from 0 on up [21] (argument extensively covered in the *background* chapter in the section on *Storage Trie*);
- the `blockId`: block identifier useful for the proof request to understand the update status of the requested information.

A request for verification is then made to the bridge contract via the function `sendRequest`, which internally calls the bridge contract function `request` and waits for a positive or negative outcome of the request, which corresponds to the receipt of a:

- `RequestServed` event in the case of successful verification;
- `BlockNotFound` event in case the block specified in the request was not found in the internal state of the contract so the verification could not be done.

## 5.4 BlockSaver

We continue with `BlockSaver.py` (related to appendix C), the first module that mimicking the behaviour of a user on C2 that periodically fetches the header data of new blocks added to *C2* and calls the bridge contract's `saveBlock` function. It does this to trigger the update of the internal state relative to the compact history of *C2*(`lightBlockchain`).

This "user" establishes both a connection to *C2* and *C1* via web3. The former to query *C2* about the information of its new blocks. The latter to retrieve a reference to the `Bridge.sol` contract deployed on *C1*.

First of all, this component asks for the address of the deployed contract on *C1* and, via the latter and the web3 connection to *C1*, it obtains a reference to the `Bridge.sol` smart contract.

Calling the `retriveNewBlock(bridgeContract)` function periodically updates the contract status related to new blocks on *C2*. This function calls the auxiliary function `RLPEncodeBlockHeader(w3,blockId)` which takes as parameters the connection via `web3` to the *C2* chain and the *id* of the block we are looking for. The latter takes care of:

1. Retrieving the data of the block identified by `blockId` via the function offered by `web3`, `get_block(blockId)`.
2. Constructing a list containing all the fields of the block header useful for calculating the block hash.
3. Returning this list in the form of *RLP encoding* [6] together with the block hash field.

RetriveNewBlock goes on by calling the `saveBlock` function of the bridge contract by passing it the pair computed by the auxiliary function.

Finally, it waits for a `NewBlockAdded` event thrown by the contract, in case the block has been successfully saved.

## 5.5 Listener

The last component we developed, also mimics a user connected to the *C2* chain; the `Listener.py` (related to appendix D). It also makes a connections via `web3` to *C1* to get a reference to the bridge contract and to *C2* to ask for information regarding the contracts stored on it.

As the *BlockSaver*, also the *Listner*, first of all, fetches a reference to the `Bridge.sol` contract via the address of the contract.

Afterwards, it remains waiting for `RequestLogged(requestId, account, key, blockId)` events, witch arise from verification requests made to the `Bridge.sol` contract.

When one of these events is raised, the *Listener*:

1. retrieves the requested information and its proof on C2 via the `web3`'s `get_proof(account,key, blockId)` function;

2. constructs a `StateProof` structure with the data just retrieved;

3. calls the bridge contract function `verify(requestId, StateProof, blockId)` that will go through the verification to assure the user on *C1* that the requested information is trustworthy.

# Chapter 6

# Experimental Evaluation

In this chapter we will present the results obtained in testing the methods offered by the `Bridge.sol` contract. We will then focus, on comparing the performances of the three proposed solutions (*basic*, *sliding window* and *array*). Depending on which one is actually used, the `Bridge.sol` contract will require more or less *gas* for its deployment and execution of the public functions it offers (`request`, `saveBlock` and `verify`). *Gas* is the unit used to measure the amount of computational power to perform some operation on the Ethereum protocol. A more in-depth explanation, can be found in [40].

## 6.1 The Test Environment

To carry out tests on the `Bridge.sol` contract, we need two Ethereum chains acting as *C1* and *C2*. These are:

- *C1*: we recall that this is the reference chain for the users requesting information from another chain. On it is also deployed the *Bridge.sol* contract. This chain was created through *Ganache* [41], an open source software that allows us to create a local blockchain in ram. The generated blockchain replicate behaviour and features of the Ethereum blockchain. Since it is a test blockchain, however, we can configure it. For example, we can make it available at `localhost` address (`http://127.0.0.1:8545`). More importantly, we have chosen to configure the chain to execute transactions instantaneously, i.e. instantaneous mining. In reality, each transaction concerning the execution of a contract will have to wait until it is added to a block in the chain to be executed, and this involves a certain amount of time that is often constant (e.g. 14 seconds on average in Ethereum). We have chosen

to eliminate this time by having blocks mined immediately as soon as a transaction is received to be able to emasure the real execution time of each function within our tests.

- *C2*: the *C2* chain is the one containing information to be retrieved. For the latter we used a real chain; these chain that would guarantee the following two characteristics:
  - real contracts with real information inside them must be stored on it. This is to give us a way to find them through the `Bridge.sol` contract and the *Listner*;
  - it must be a chain that is updated periodically through new blocks. Preferably, the period between the addition of one block and the next must be short (on the order of seconds or minutes) so that the `Bridge.sol` can be tested quickly with the *BlockSaver*.

  So we exploited the Ethereum main chain through the use of **QuikNode** [42]. The latter is a tool that provides an *https address* that allows to connect to the Ethereum main chain.

## 6.2 Testing

Using a local chain as chain *C1* and Ethereum main net as chain *C2*, the `Bridge.sol` contract was executed to assay its proper functioning. The most important metrics of evaluation concerning smart contracts is the *gas* consumption as a result of their execution.

Let's start by evaluating the deployment of the `Bridge.sol` contract which naturally differs according to the three different implementations. This is because the calculation of the *gas* consumed is a deterministic operation. The consumed gas varies as the operations performed within the execution of a contact change. Therefore, the three implementations that partly perform different operations in their execution also differ in the amount of gas consumed.

### 6.2.1 Deployment

Before analysing the deployment cost in gas, let us explain how gas consumption and its price works on the Ethereum blockchain.

### Ethereum Gas Price

In Ethereum, each transaction is a kind of program that instructs the Ethereum Virtual Machine (EVM) about the actions to be performed to move funds or execute smart contracts. Each of these transactions has a cost, a kind of "tax" that must be paid to the miners to incentive them to run the network. Ethereum's developers, however, have decided to assign a constant value to the different transactions that can be performed on the platform. This, in order to avoid large fluctuations in the price of transactions as the value of ETH, Ethereum's trademark cryptocurrency, rises or falls. Therefore, each individual transaction will have a predetermined value in Gas, corresponding to the computational cost of the transaction itself. To carry out any transaction such as a transfer of funds, 21000 Gas is required. This amount should be added to the costs for the operations required to interact with the smart contracts. For example, an ADD (addition of two integers) transaction requires 3 unit of gas.

Each unit of Gas has a price in *Gwei* ($1 * 10^{-9} ETH$), which is given by the supply and demand for transactions in Ethereum. Specifically, the user is able to choose the value to pay for the amount of Gas needed to perform the transaction, and if a miner accepts it, it will be executed. In turn, supply and demand, depend on how much other Ethereum users are willing to pay for a unit of Gas. So, it is important to remember, that this price is highly variable and in a few days can double or halve.

However, there are some references that can help you make a fairly accurate estimate of how much you can spend on confirming transactions. The *safelow* is the minimum price per unit of Gas that assures that a transaction will be successful. In this regard, at least 5% of the network hash power must accept this price for Gas. Currently (January 2023), this cost is equivalent to 22 Gwei per Gas unit. If a user opts for *safelow*, they will enjoy reduced transaction costs, but the transaction, may will remain pending for longer (maximum 10 minutes).

The *standard* price, on the other hand, is the price currently accepted by the *top miners* (i.e. miners who take part in the execution of, at least, 50% of the transactions). This price also varies widely, and in general, since it offers faster execution of transactions, it stands at a higher price than *safelow* price.

Returning now to the analysis related to the deployment of the contract `Bridge.sol`, the table below shows gas consumption based on the implemented solutions:

| Solution | Deployment Consumption | ETH Consumption | USD |
|---|---|---|---|
| Basic | 2044028 gas unit | 0,045 | 69,53 |
| Sliding Window | 2115142 gas unit | 0,047 | 72,62 |
| Array | 2174516 gas unit | 0,0478 | 73,85 |

Table 6.1: Usage of gas units for deployment

As can be seen, the usage of more variables in the internal state of the last two solutions, resulted in an increase in the gas used for deployment.

The table 6.1 show also the actual USD cost of deployment of the contract `Bridge.sol`. The calculations of the values in the table were made based on the *safelow* gas price in Gwei in January 2023. We recall, this price is equal to an average price of 22 Gwei per unit of gas.

The cost in ETH is calculated according to the following formula:

$$ETH\ Consumption = Deployment\ Consumption * Safeslow\ Price * (10^{-9}) \tag{6.1}$$

Conversion to USD, on the other hand, is calculated according to the following formula based on an average price for an ETH of 1545 USD:

$$USD = ETH\ Consumtion * ETH\ price\ in\ USD \tag{6.2}$$

To spend less on deployment, an alternative would be to run the transaction on Polygon's Layer 2 chain (discussed in the chapter on related work) instead of Ethereum's main chain. Indeed, Polygon allows cheaper and faster transactions using Layer 2 side-chains. Its standard price is 288 Gwei per unit of gas and its currency called *MATIC*, as for ETH is $1 * 10^9$ Gwei. The savings, however, are in relation to the fact that unlike ETH an MATIC costs only 1,28 USD. In addition, the time needed to have a transaction executed ranges from 30-60 seconds for a *standard* transaction to 5-10 seconds for a *rapid* transaction. Obviously, the price for a *rapid* transaction is higher than the price for a *standard* transaction.

### 6.2.2 Request

We now consider the `request` function. Since it is implemented in the same way in all the three solutions, there is no need to differentiate them. The `request` function lends itself on a gas usage of 101540 for requests other

than the first. The first call of the function after contract deployment, uses an amount of gas ranging between 131540 and 151540 probably due to the instantiation of data structures useful for the function.

### 6.2.3 saveBlock

Let us now analyse the `saveBlock` function. We recall that, this is the method that will potentially be called several times i.e. at each update of the *C2* blockchain. In addition, it is the function whose internal logic varies the most for the three proposed solutions. We begin our analysis by examining the gas consumption and time spent for each call, related to the three solutions (*basic*, *sliding window*, *array*).

#### basic

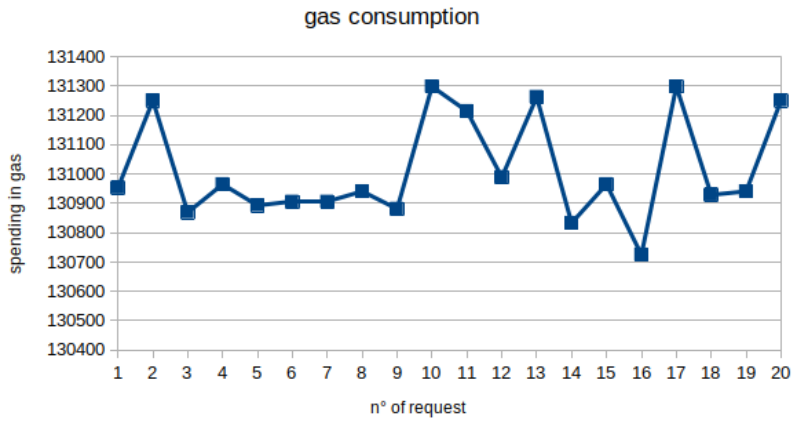


Figure 6.1: units of gas spent for each of the first 20 calls of the *basic* solution.

As we can see from the previous figure, for the *Basic* solution, the consumption does not vary much between the first calls and the subsequent ones since the `saveBlock` function always behaves the same way in this case. The use of gas for different calls, ranges between 130700 and 131300 units.

Turning instead to the time required to complete a `saveBlock` request to the *basic* Bridge contract. We say that time required is defined as the seconds between the *saveBlock* request to the *basic* bridge contract and the receipt of the `NewBlockAdded` event by the user who made the request. Below there is a graph showing the time spent on the first 20 saveBlock calls to the *basic* bridge contract.

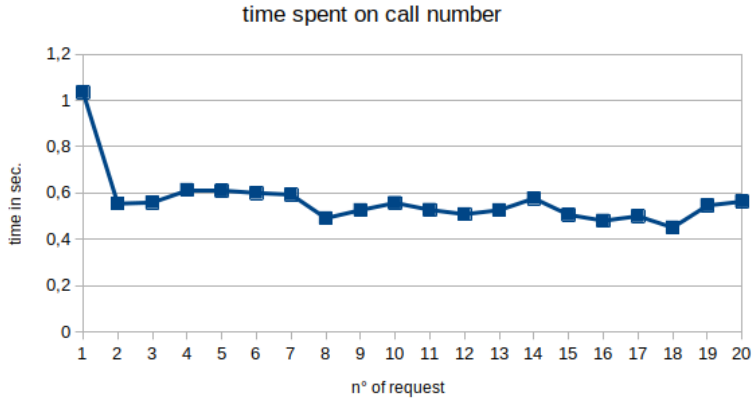


Figure 6.2: Seconds spent on each call in `saveBlock` *basic* solution

In the graph 6.2, there are seconds spent per request on the y-axis. On the x-axis are the various requests from the first to the twentieth identified by an index from 1 to 20. From the figure 6.2 it is clear that time spent is fairly constant, ranging from 0.4 to 0.6 seconds per call. The only exception is the first call, which costs approximately 1 second, probably because of the time spent accessing and allocating new memory space for the internal variables needed for the `saveBlock` method. Considering, from the data collected, an average of 0.54 seconds spent to serve a saveBlock call, the *basic* bridge contract will be able to serve almost 2 saveBlock calls per second.

**sliding window**

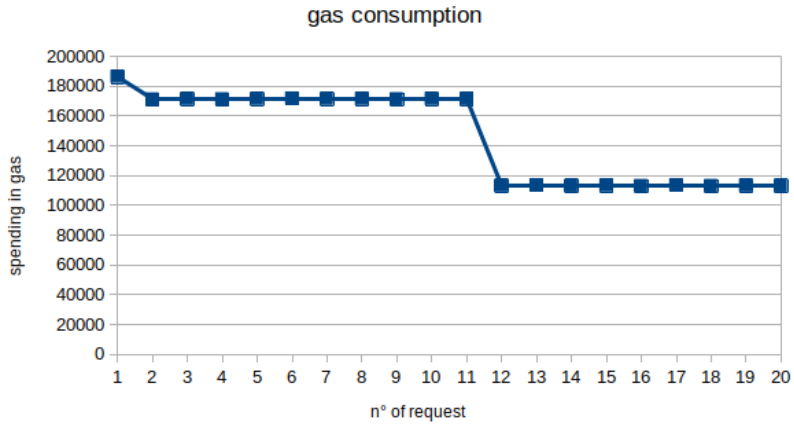


Figure 6.3: units of gas spent for each of the first 20 calls of the *sliding window* solution.

In contrast to the *basic* solution, in the *sliding window* solution, gas consumption is highly variable. As can be seen from the figure 6.3 the amount of gas units spent per call will form 3 distinct steps. These go to differentiate the first call, the next ten and the following where we enter into a kind of "steady state". The number of requests needed to enter this "steady state" depends from the size (`windSiz`) of the window. If this *size* is for example, equal to 10 (as in the figure 6.3), the next ten calls after the first one will all have a similar cost that is less than that of the first call but still greater than that of the subsequent calls. These consumption are around 190000 gas units for first call, around 170000 for each of the next ten calls and around 113000 for the each call in the "steady state". Generally, the first call and the next ten calls will be more expensive because they are the ones that will first access new memory resources and data structures offered by the contract. After the first eleven calls, assuming as a basis that our`windSize` is always 10, the `saveBlock` function changes behaviour and enters what may be called *steady state*. In this state, the `saveBlock` function does not access any new memory locations but always reuses the same ones previously accessed as the total number of blocks saved in the compact history is greater than 10 so, for each call one block is deleted from the compact history and another is added. Reusing the same memory cells leads to less consumption in terms of gas units used.

Switching to the seconds spent per call of the `saveBlock` function with *sliding window* we show the related graph below.

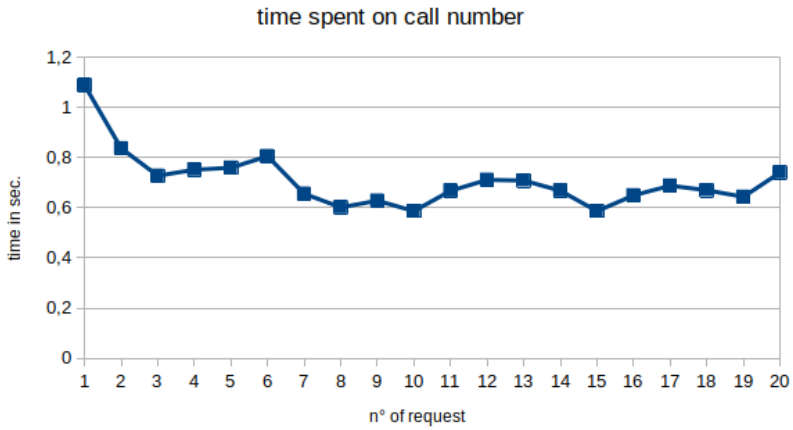


Figure 6.4: Seconds spent on each call in `saveBlock` with *sliding window*

Figure 6.4 shows the cost in time of this *sliding window* solution. Ignoring the time cost of the initial call (always afflicted by the same problems previously discussed), we can observe that the seconds spent per call range from a maximum of 0.83 seconds to a minimum of 0.58. Therefore, the range of results, as well as the average seconds which are around 0.70 are greater than the *basic* solution. This is easily justified by the fact that, having a more complex logic, compared to the *basic* solution, the number of single instructions to be executed are greater. The number of calls executed per second therefore stands at 1.43.

**array**

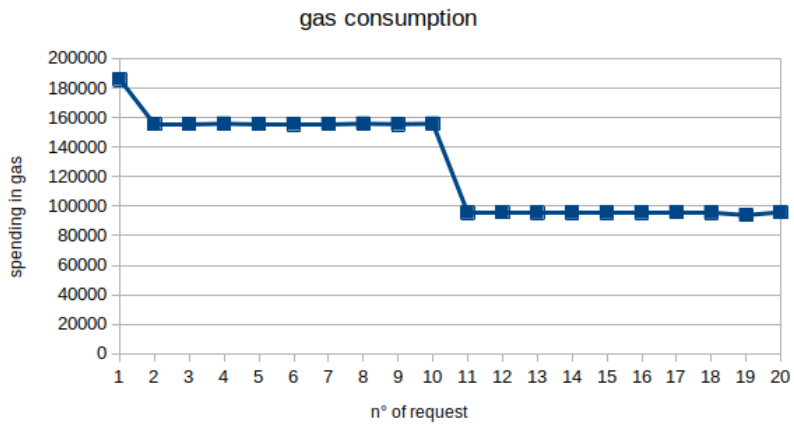


Figure 6.5: units of gas spent for each of the first 20 calls of the *array* solution.

As with the *sliding window* solution, for the *array* solution, as we can see from the figure 6.5, we will have three stairs of consumption. In fact, also in this case, the width of these steps will depend on the *size* chosen for the array according to `arrSize`. In figure 6.5 we see that the first call will cost around 185000 units of gas, the next 10 calls (for an `arrSize` equal to 10) will cost around 155000 units of gas per call while the "steady state" calls around 95000 units of gas per call. The reason for these stairs in the chart of gas consumed, is to be found in the same reasons mentioned for the *sliding window* solution. That is, the first calls are more expensive since they are the first to use memory resources and contract data structures. The "steady state" calls are the least expensive since, especially in the case of the array solution, they will always reuse the same memory locations. Arrays, in addition, can also be saved in a single slot in the storage tree.

As with the previous cases, after discussing the gas consumption used for the `saveBlock` call, let's move on to the seconds spent on each call in the *array* solution.

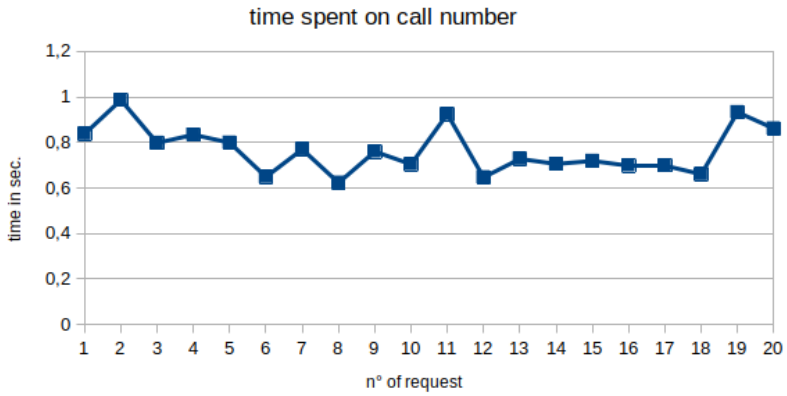


Figure 6.6: Seconds spent on each call in `saveBlock` with *array*

From the previous figure (6.6) we can see that the *array* solution is the most expensive in terms of seconds spent per call. Indeed, on average, the seconds spent on a `saveBlock` call with array are 0.77. The number of `saveBlock` calls, which the contract will be able to perform per second, will be 1.3. The reason for this increase, in seconds spent, compared to other solutions is to be found in the adoption of the array structure instead of the map structure. This is because, managing an array at the instruction level costs more in terms of time than managing a map.

Below we show the table with the comparison of average gas usage related to different solutions.

| Solution | *avg(IC)* | *avg(SSC)* |
|---|---|---|
| Basic | 133400 gas unit | 131016 gas unit |
| Sliding Window | 171450 gas unit | 113256 gas units |
| Array | 155656 gas unit | 95454 gas unit |

Table 6.2: Usage of gas units for `saveBlock` calls. *IC* : Initial Consumption. *SSC* : Steady State consumption

From 6.2 we can see that, solutions other than *basic* initially incur a higher cost since they use and allocate more state variables for their operation. At the same time, however, when we get to ”steady state”, the calls of the

solutions other than *basic* perform better. In particular, the *sliding window* solution saves about 20000 gas units per call compared to the *basic* solution. Ultimately, however, the solution with *array* performs the best. For each call in "steady state", it manages to save about 40000 gas units compared to the *basic* solution and about 20000 gas units compared to the solution with *sliding window*.

Turning now to the consumption of seconds, below we find the summary table that compares the three solutions in relation to the results regarding the seconds spent per call.

| Solution | *min(ss)* | *max(ss)* | *avg(ss)* | calls x second |
|---|---|---|---|---|
| Basic | 0.4 | 0.6 | 0.54 | 1.85 |
| Sliding Window | 0.58 | 0.83 | 0.70 | 1.43 |
| Array | 0.64 | 0.98 | 0.77 | 1.3 |

Table 6.3: Seconds spent for `saveBlock` calls. *ss* : seconds spent

From the results of the previous table 6.3 we can see that in terms of seconds spent per call the best solution is the *basic* one. Then the other two follow with slightly better results for the *sliding window* solution. However, since, we know that on average a block is added to an Ethereum-based chain every 15 seconds, all three solutions are more than sufficient to handle the constant updating of the compact history.

It is interesting to see, in relation to the two previous tables (6.2, 3.1) that there is an inverse relationship between gas spent and seconds spent per call. In relation to the needs (i.e. use of gas and seconds spent) it will therefore be necessary to choose the solution that provides the most advantageous trade-of.

### 6.2.4 verify

We complete the discussion on experimental evaluations, with the evaluation of the `verify` function. Although not dependent on the solution used since, the operations performed in the verify function itself do not change for the three solutions, the verification function is the one most subject to variation in gas usage. In fact, from the tests carried out it appears that it is dependent on the amount of nodes present in the two types of proof to be

verified, namely `account Proof` and `storage Proof`. This happens because the verification procedure consists of comparisons between nodes in a trie and so naturally the deeper the trie the longer the verification will be.

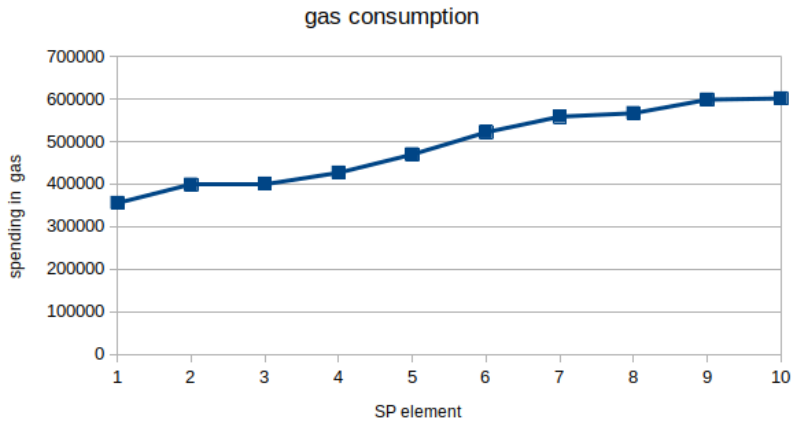


Figure 6.7: Gas consumption for a `verify` request related to the number of elements in the *Storage Proof*

.

The minimum possible gas used is when the `storage Proof` bound to the required `key` is empty. In fact, in this case, only the `account Proof` needs to be verified, which, in terms of gas, costs about 300000 units. The `account Proof` often contains as many as 8 or 9 elements but with elements as long as 532 characters. If there is also to be verified the `storage Proof`, the cost of the whole function grows to 600000 gas units. This cost is in the case where we need to perform storage verification, for example, on a proof of 10 elements of length varying between 100 and 300 characters.

### 6.2.5 cross-chain data transfer

Finished the evaluations regarding individual methods, we focus on one of the most significant analyses regarding our proposal. The question we want to answer is: how long, on average, will a user on *C1* have to wait to retrieve data from *C2* as the number of requests to the `Bridge.sol` contract increases?

Of course, just as the gas spent on the `verify` function, the waiting time for each *verify* is also dependent on the depth of the merkle tree constructed through the nodes in the *Storage Proof*. Since we are interested in an average time, we will average between the various times found. These times range

from a minimum of about 5 seconds for verification on a 3-element *Storage Proof* to a maximum of about 8 seconds for verification on larger *Storage Proofs*. On average, `Bridge.sol` will spend 7.5 seconds to return a *C2* information to a user on *C1* . As the number of requests received at the same time increases, we show the average waiting time in the following graph.

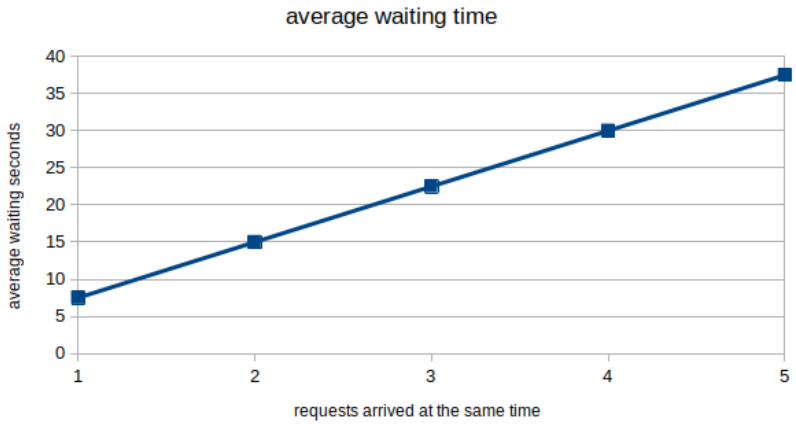


Figure 6.8: Average waiting time per request as the number of requests arrived at the same moment increases

.

From the graph 6.8 we see that the average seconds of waiting time for each request grows proportionally to the number of requests arriving at the same time. To keep this wait low it will be necessary to create multiple instances of `Bridge.sol` working at the same time. The instances needed will be at least equal to the number of requests that on average arrive per second to keep the wait to the average 7.5 seconds.

To scale the time further (to halve it), two contracts could be created to go to work simultaneously on a request to the `Bridge.sol` contract. One of them will only verify the *Account Proof* related to the request. The other will only verify the *Storage Proof*.

# Chapter 7

# Conclusions and Future Work

The goal of this work was to address the problem of trusted cross-chain information exchange, and, for this purpose, an attempt was made to exploit existing technologies in a new context. The technology chosen was Authenticated query systems, which are typically related to the light/full node communications of a blockchain protocol. We proposed CARONTE, a cross chain communication protocol where all metadata related to the performed verification and information exchange remain saved on the blockchain on which the bridge contract is deployed. The advantage of employing a bridge contract based solution, as CARONTE, relies on the trustworthiness of smart contracts execution, so that no additional trust requirements are imposed on the participants. Our proposal is therefore a hybrid of multiple technologies, trying to exploit their best features, to provide an alternative that has not yet been explored in the field of blockchain scaling solutions. Indeed, it is part of the large family of Layer 2 solutions trying to scale the main chain by allowing information exchanges between side-chains or side-chains and the main chain.

After defining the characteristics and structure of CARONTE, we have discussed the actual development of the bridge contract at its core. Initially, we developed a basic first implementation, and then we improved it through a set of optimizations. We have defined two optimized solutions, aimed at improving the contract performances with respect to gas consumption per call.

To test the bridge contract and evaluate its performances, three further modules called `requester`, `blockSaver`, and `listener` were developed. Once everything needed to start the tests was set, the bridge contract was contacted to verify its functioning. The results of the tests, with respective evaluations, largely reflect our expectations in terms of gas consumption and the latency aimed for when structuring the protocol.

However, considering the obtained results, many other improvements can still be applied. Let us now examine those which, according to our evaluation, could be more impactful for expanding the possibilities of use and the services offered by CARONTE.

- Integration of **zero knowledge proofs**. In line with the philosophy of using technologies that are already well known but used in other areas, we could expand CARONTE to offer verification according to the zero knowledge model. As discussed in the relative section on related works, the zero knowledge model is widely used in rollup solutions [4]. It deals with the exchange of information between different actors of a protocol, but without any of them necessarily having to reveal in clear the actual value of the information exchanged. According to this model it is possible to add to the methods offered by `Bridge.sol` a further possibility of verifying the data exchanged according to a *zero knowledge proof*. Specifically, the *ZoKrates* framework [37] can be used to implement this feature. Using this model, the information on source chain (*C2*) need not be disclosed to the destination chain (*C1*), but rather teh result of a function applied to it. This scenario further enforces the privacy of using CARONTE with private chains as source chain (*C2*).

- improvement of the techniques for verifying the headers of *C2* while updating its compact history in the `Bridge.sol` contract. We have previously explained that, currently, our proposed bridge contract correctly checks for *C2* headers integrity. But our control process suffers from a very obvious weakness. Fake, but still valid, headers can in fact be created by malicious potential blockSaver users on *C2*. Actually, if a fake header is created ad-hoc and accompanied by a hash calculated accordingly to the previous steps, our verification process would consider it acceptable. The header will then be added to the compact history in `Bridge.sol` even though it does not really exists on *C2*. This, unfortunately, is definitely a vulnerability that was deliberately left open, since our main goal was to build a reliable protocol, but, for now, only with respect to verifying the information to be exchanged. To eliminate this weakness, a solution may include a verification proof also in the *C2* compact history update procedure in `Bridge.sol`. A concrete solution that could be taken as an example is the one discussed in the recently published paper [43]. The authors of the paper propose a protocol very similar concept to CARONTE, but regarding the verification of chain headers via zero knowledge proofs.

- use a **cryptographic accumulator**[44] to save *C2*'s compact history in the bridge. A cryptographic accumulator is a one-way membership cryptographic function efficient in both space and time. It allows users to certify that potential candidates are members of a certain set without revealing the individual members of the set. This structure, unlike the other structures examined for this task, offers a constant cost in space. Such a tool could be used in CARONTE to save *C2*'s compact history in constant space, with a huge saving in blockchain space, at the expense of the added complexity in the membership verification and retrieval operations, that would require more complex cryptographic operations.

# Appendix A

# Bridge.sol with array

```
1  pragma solidity 0.6.0;
2  pragma experimental ABIEncoderV2;
3
4  import{StateProofVerifier} from "Contracts/StateProofVerifier
       .sol";
5  import {RLPReader} from "Contracts/RLPReader.sol";
6
7  contract Bridge {
8
9      using RLPReader for RLPReader.RLPItem;
10     using RLPReader for bytes;
11
12     struct BlockHeader{
13         bytes32 hash;   //hash of the block used to verify
               its correctness
14         bytes RLPHeader; //rlp encoding of the block header
               to be checked
15     }
16
17     struct StateProof {
18         address account; // Address of the C2 contract
19         bytes [] accountProof; // Proof that certifies the
               existence of the contract
20         bytes32 storageRoot; // root of the storage trie for
               account
21         bytes key; // position in slots
22         bytes value; // value stored at key slot
23         bytes [] storageProof; // Proof that certifies the
               existence of the variable inside the contract
24     }
25
26     struct Request {
27     address account; // Address of the C2 contract we want to
```

```
        read data from.
28      uint key; // Numeric identifier of the C2 variable we
            want to read.
29      uint blockId; // Identifier of the C2 block we want to
            read data from.
30      uint date; // Timestamp of the request.
31      bool served;  // Equals true if the request has been
            served, false otherwise.
32      uint256 response; // The value associated with the
            variable.
33      }
34
35      //internal state
36      // Counts the total number of requests.
37      uint private requestCounter;
38      // Counts the number of served requests.
39      uint private servedCounter;
40      //arrived requests
41      Request[] private requests;
42
43      uint arrSize; //size for the lightBlockchain array
44      uint blockCounter = 0; //conter which counts the number
            of C2 blocks arrived in the bridge contract
45      uint lastKey = 0; //variable that save the block number
            of the last block saved
46      StateProofVerifier.BlockHeader[] private lightBlockchain;
            //array for save the block of C2
47
48      constructor(uint _arrSize) public {
49          arrSize = _arrSize;
50          //initialize to 0 the arrSize position of
                lightBlockchain
51          for(uint i = 0; i < _arrSize; i++){
52              lightBlockchain.push(StateProofVerifier.
                    BlockHeader(0,0,0,0));
53          }
54
55      }
56
57      /**
58      *  @dev if the blockHeader passed pass the verification
            the block is saved in the contract
59      *  @param _blockHeader the structure that contain the
            hash of the blockHeader and the header's RLP encode
60      *  @return the identifier of the blockHash saved.
61      */
62      function saveBlock(BlockHeader memory _blockHeader )
            public returns (uint) {
63
```

```
64          //chack that the block to be added is ok
65          StateProofVerifier.BlockHeader memory bHeader =
                verifyBlockHeader(_blockHeader);
66
67          uint toDelate = 0;
68          //if the block that you are trying to save in
                lightBlockchain is newer than the one in counter%
                arrSize psition
69           if(bHeader.number >
70              lightBlockchain[blockCounter%arrSize].number){
71               //save in toDelate variable the that position
72               toDelate =
73               lightBlockchain[blockCounter%arrSize].number;
74               //update the array in that position with the
                     new block
75               lightBlockchain[blockCounter%arrSize] =
76               bHeader;
77               //update blockCounter
78               blockCounter = blockCounter +1 ;
79               //update lastKey
80               lastKey = bHeader.number;
81           }
82
83          emit NewBlockAdded(bHeader.hash, bHeader.
                stateRootHash, bHeader.number, bHeader.timestamp,(
                blockCounter-1)%arrSize, toDelate);
84
85          return  lightBlockchain[blockCounter%arrSize].number;
86      }
87
88      /**
89      *  @dev This event is emitted whenever a new request is
            created
90      *  and saved inside the contract.
91      */
92      event RequestLogged(
93          uint indexed requestId, // Request identifier
94          address account, // Address of C2 contract
95          uint key, // Numeric identifier of the variable
96          uint blockId // Identifier of C2 block
97      );
98
99      /**
100     *  @dev This event is emitted whenever a logged request
            is served by the verify function
101     */
102     event RequestServed(
103         uint indexed requestId, // Request identifier
104         address account, // Address of C2 contract
```

```
105         uint key, // Numeric identifier of the variable
106         uint blockId, // Identifier of C2 block
107         uint256 reply // Response received by C2 node
108     );
109 
110     /**
111     *  @dev This event is emitted whenever a new header pass
        the verification
112     *  and is added to the lightBlockchain of the contract
113     */
114     event NewBlockAdded(
115         bytes32 hash,
116         bytes32 stateRootHash,
117         uint256 number,
118         uint256 timestamp,
119         uint256 pos,
120         uint256 delblock
121     );
122 
123     /**
124     *  @dev This event is emitted whenever a header not pass
        the verification
125     *  and so the request is rejected by the contract
126     */
127     event BlockNotFound(
128         uint256 number
129     );
130 
131     /**
132     *  @dev This data structure keeps track of all requests.
133     *  NOTICE: the id of a request coincides with its index
        in the array.
134     */
135     function getTotal() public view returns (uint) {
136         return requestCounter;
137     }
138 
139     /**
140     *  @return the total number of requests served by the
        contract
141     */
142     function getServed() public view returns (uint) {
143         return servedCounter;
144     }
145 
146     /**
147     *  @return the number of pending (i.e., not served)
        requests.
148     */
```

```
149     function getPending() public view returns (uint) {
150         return requestCounter-servedCounter;
151     }
152 
153     /**
154     *  @param id identifier of the request
155     *  @return the request with the specified identifier
156     */
157     function getRequest(uint id) public view returns (Request
            memory) {
158         // Check if the supplied index is legal.
159         require(0 <= id && id < requests.length,
160         "Error: invalid request id");
161         return requests[id];
162     }
163 
164     /**
165     *  @param blockId identifier of the required blockHeader
166     *  @return the blockHeader of the specified blockId
167     */
168     function getBlock(uint blockId) public view returns (
            StateProofVerifier.BlockHeader memory) {
169         return lightBlockchain[blockId];
170     }
171 
172     /**
173     *  @param _account of the contract on C2 of which we want
            the information
174     *  @param _key index of the information in the contract
            on C2
175     *  @param _blockId identifier of the block that represent
            the update status of the requested variable
176     *  @return the identifier of the request that was taken
            in charge.
177     */
178     function request(address _account, uint _key, uint
            _blockId) public returns (uint) {
179         Request memory r;
180         uint requestId = requestCounter;
181         r.account = _account;
182         r.key = _key;
183         r.blockId = _blockId;
184         r.date = block.timestamp;
185         r.served = false;
186         requests.push(r);
187         emit RequestLogged(requestId, _account, _key,
                _blockId);
188         requestCounter++;
189         return requestId;
```

```
190     }
191
192     /**
193     *  @dev check if the block is present in contract's
            lightBlockchain
194     *  @param _requestId id of the request that need for save
             the request as rejected if the blockId is not present
             in lightBlockchain
195     *  @param _blockId identifier of the block that represent
             the update status of the requested variable
196     *  @return true if the block is saved false otherwise
197     */
198     function blockIsPresent(uint _requestId, uint _blockId )
            public returns (bool){
199
200         //verify that the requested block is present in
               contract's lightBlockchain
201         bool isPresent = true;
202         //difference between last saved block and the numer
               of the block required by the verificatio
203         uint diff = lastKey - _blockId;
204
205         //if lastKey < _blockId  (and so the required block
               is bigger than the last inserted block in the
               lightblockchain) or
206         //the required block is too outdated  or
207         //the required block is in the range of the arraySize
                but not saved in the lightBlockchain (missing
               block)
208         if(diff < 0  || diff >= arrSize ||
209            lightBlockchain[((blockCounter-1)%arrSize)-diff].
                  number != _blockId){
210             emit BlockNotFound(_blockId);
211             isPresent = false;
212             //The request is saved as served but rejected
                   with response = 0
213             requests[_requestId].served = true;
214             requests[_requestId].response = 0;
215         }
216
217         return isPresent;
218     }
219
220     /**
221     *  @dev verification for block header.
222     *  @param _blockHeader the structure that contain the
            hash of the blockHeader and the header's RLP encode
223     *  @return the parsed header if it pass the check
224     */
```

```
225     function verifyBlockHeader (BlockHeader memory
            _blockHeader )
226                 public returns (StateProofVerifier.
                        BlockHeader memory){
227
228         //parse the block header
229         StateProofVerifier.BlockHeader memory header =
230          StateProofVerifier.parseBlockHeader(_blockHeader.
                RLPHeader);
231
232         //checks that the hash field passed as a paramenter,
                matches the has computed via kekka256 of the RLP
                encoding
233         require(_blockHeader.hash == header.hash,
234                 "blockhash mismatch");
235
236         return header;
237     }
238
239     /**
240     *  @dev verification for account proof
241     *  @param _stateProof the structure that contain all the
            fields arrived from the C2 chain proof request
242     *  @param _blockId identifier of the block that represent
             the update status of the requested variable
243     *  @return the parsed account if it pass the check
244     */
245     function verifyAccountProof(StateProof memory _stateProof
            , uint _blockId)
246                 public returns (StateProofVerifier.Account
                        memory){
247
248         //parse the account proof
249         RLPReader.RLPItem[] memory accountProof =
                parseProofToRlpReader(_stateProof.accountProof);
250         //difference between last saved block and the numer
                of the block required by the verification
251         uint diff = lastKey - _blockId;
252         //verify the account proof
253         StateProofVerifier.Account memory account =
254          StateProofVerifier.extractAccountFromProof(
255             keccak256(abi.encodePacked(_stateProof.account)),
256             lightBlockchain[((blockCounter-1)%arrSize)-diff].
                    stateRootHash,
257             accountProof);
258         return account;
259     }
260
261     /**
```

```
262     *  @dev verification for storage proof
263     *  @param _stateProof the structure that contain all the
            fields arrived from the C2 chain proof request
264     *  @return the value searched if the storage proof pass
            the check
265     */
266     function verifiyStorageProof(StateProof memory
            _stateProof)
267        public returns (StateProofVerifier.SlotValue memory){
268
269         //parse the storage proof
270         RLPReader.RLPItem[] memory storageProof =
               parseProofToRlpReader(_stateProof.storageProof);
271
272         //verify the storage proof
273         StateProofVerifier.SlotValue memory slotValue =
274          StateProofVerifier.extractSlotValueFromProof(
275              keccak256(abi.encodePacked(_stateProof.key)),
276              _stateProof.storageRoot,
277              storageProof);
278         return slotValue;
279     }
280
281     /**
282     *  @dev verification for all the state proof
283     *  @param _requestId the id of the request that we serve
            with this function
284     *  @param _stateProof the structure that contain all the
            fields arrived from the C2 chain proof request
285     *  @param _blockId identifier of the block that represent
             the update status of the requested variable
286     *  @return true If all verifications are passed
287     */
288     function verify(uint _requestId, StateProof memory
            _stateProof, uint _blockId )
289         public returns (bool){
290
291         //check that the block[_blockId] is present in the "
               lightBlockchain" of the contract
292         if(blockIsPresent(_requestId, _blockId) == true){
293             //check account proof
294             StateProofVerifier.Account memory account =
                   verifyAccountProof(_stateProof, _blockId);
295
296             //check storage proof
297             StateProofVerifier.SlotValue memory slotValue =
                   verifiyStorageProof(_stateProof);
298
299             //The proof is accepted: first we record this
```

```
                fact on the blockchain.
300             requests[_requestId].served = true;
301             requests[_requestId].response = slotValue.value;
302
303             // Then we trigger a 'RequestServed' event to
                    notify all possible listeners.
304             emit RequestServed(
305                 _requestId,
306                 requests[_requestId].account,requests[
                        _requestId].key,
307                 requests[_requestId].blockId,
308                 requests[_requestId].response);
309             return true;
310         }
311         return false;
312     }
313     /**
314     *  @dev auxiliary function that parse a generic proof in
            a list of RLP encode
315     *  @param _proof list that rapresent the proof for the
            requested value
316     *  @return the proof in the RLPItem list form
317     */
318     function parseProofToRlpReader(bytes[] memory _proof)
            internal pure returns (RLPReader.RLPItem[] memory){
319         RLPReader.RLPItem[] memory proof = new RLPReader.
                RLPItem[](_proof.length);
320         for (uint i = 0; i < _proof.length; i++) {
321             proof[i] = RLPReader.toRlpItem(_proof[i]);
322         }
323         return proof;
324     }
325 }
```

# Appendix B

# requester.py

```
1  from web3 import Web3,HTTPProvider
2  from solcx import compile_source
3  import time
4
5  def compile_source_file(file_path):
6
7      with open(file_path, 'r') as f:
8          source = f.read()
9    #returns the [abi,bin] tuple for the first element of the
       dictionary, i.e. the data related to Bridge
10     return compile_source(source,
11                           output_values=["abi", "bin"],
12                           solc_version="0.6.0")
13                             .get('<stdin>:Bridge')
14
15 def deploy_contract(w3, contract_interface):
16   #if you want to use bridge with sliding window or array put
       size parameter in the constructor() call function.
17         tx_hash = w3.eth.contract(
18                         abi=contract_interface['abi'],
19                         bytecode=contract_interface['bin'])
20                             .constructor().transact()
21         address = w3.eth.get_transaction_receipt(tx_hash)
22                     ['contractAddress']
23         return address
24
25 #web3 for take the reference to the contract deployed on c1
26 chainAddress = 'http://127.0.0.1:8545'
27 web3 = Web3(HTTPProvider(chainAddress))
28 web3.eth.defaultAccount = web3.eth.accounts[0]
29
30
31
```

```
32 def sendRequest(bridgeContract,_proofAddress,_key,_blockId):
33     #send a verification request to the contract
34     txt_hash = bridgeContract.functions.request(
35      _proofAddress, int(_key), int(_blockId)).transact()
36     txn_receipt =
37         web3.eth.wait_for_transaction_receipt(txt_hash)
38     print(txn_receipt)
39 
40     # Listen for the reply and the result.
41     requestId =
42     (bridgeContract.functions.getTotal().call())-1
43     event_filter_good =
44     bridgeContract.events.RequestServed.createFilter(
           fromBlock='latest', argument_filters={'requestId':
           requestId})
45     event_filter_bad =
46     bridgeContract.events.BlockNotFound.createFilter(
           fromBlock='latest')
47 
48     received = False
49     #waiting for the event RequestServed or BlockNotFound
50     while not received:
51                 entries_good = event_filter_good.
                       get_new_entries()
52                 entries_bad = event_filter_bad.
                       get_new_entries()
53                 if (len(entries_good) > 0 or len(entries_bad)
                        > 0 ):
54                     if(len(entries_good) > 0 ):
55                         print(entries_good)
56                     else:
57                         print(entries_bad)
58 
59                     received = True
60                 else:
61                     time.sleep(3)
62 def main():
63 
64     #compile contract and initialize it
65     contract_interface = compile_source_file('Contracts/
           Bridge.sol')
66 
67     #deploy contract
68     contractAddress = deploy_contract(web3,
           contract_interface)
69     print('requester has started!\nAddress\t: {}'.format(
           contractAddress))
70     bridgeContract = web3.eth.contract(address =
           contractAddress, abi=contract_interface["abi"])
```

```
71
72      print('usage:\n 1) address for the proof\n
73                     2) key for the proof\n
74                     3) block id')
75      while True :
76          _proofAddress = input()
77          _key = input()
78          _blockId = input()
79          sendRequest(bridgeContract, _proofAddress,_key,
                _blockId)
80
81  if __name__ == "__main__":
82      main()
```

# Appendix C

# blockSaver.py

```
1  from web3 import Web3,HTTPProvider
2  from hexbytes import HexBytes
3  from rlp import encode
4  from solcx import compile_source
5  import time
6
7  def compile_source_file(file_path):
8
9      with open(file_path, 'r') as f:
10         source = f.read()
11   #returns the [abi,bin] tuple for the first element of the
        dictionary, i.e. the data related to Bridge
12     return compile_source(source,
13                           output_values=["abi", "bin"],
14                           solc_version="0.6.0").
15                             get('<stdin>:Bridge')
16
17 #w3 for retrive the block of c2 (in this example is the
     etherium blockhain) and save they as a Lightweight
     blockchain on the contract
18 w3 = Web3(HTTPProvider('https://evocative-stylish-isle.
     discover.quiknode.pro/6
     ad754e3368653a2665e62db8659b9f179a3ae43/'))
19
20 #web3 is the Web3 reference to the chain C1 for take a
     reference to the bride contract
21 chainAddress = 'http://127.0.0.1:8545'
22 web3 = Web3(HTTPProvider(chainAddress))
23 web3.eth.defaultAccount = web3.eth.accounts[0]
24
25
26
27
```

```
28
29
30 #auxsiliary function that retrive the block header and code
       it in RLP encode
31 def RLPEncodeBlockHeader(w3,blockId):
32
33     block = w3.eth.get_block(blockId)
34     blockHeader = [
35         block.parentHash,
36         block.sha3Uncles,
37         HexBytes(block.miner),
38         block.stateRoot,
39         block.transactionsRoot,
40         block.receiptsRoot,
41         block.logsBloom,
42         HexBytes(hex(block.difficulty)),
43         HexBytes(hex(block.number)),
44         HexBytes(hex(block.gasLimit)),
45         HexBytes(hex(block.gasUsed)),
46         HexBytes(hex(block.timestamp)),
47         block.extraData,
48         block.mixHash,
49         block.nonce,
50         #HexBytes(hex(block.baseFeePerGas)) #parameter added
               to the header between the block 12000000 and
               13000000
51         #block.size
52         #block.totalDifficulty
53     ]
54
55     #check that need for the new type of header with
           baseFeePerGas field
56     if len(block) != 20:
57         blockHeader.append(HexBytes(hex(block.baseFeePerGas))
               )
58
59     for i in range(0,len(blockHeader)):
60             if blockHeader[i] == HexBytes("0x0") or
                   blockHeader[i] == HexBytes("0x00"):
61                 blockHeader[i] = HexBytes("0x")
62
63     blockHeader = [
64         block.hash,
65         HexBytes(encode(blockHeader))
66     ]
67
68     return blockHeader
69
70
```

```
71
72
73  #infinite loop that call saveBlock contract's funcion for
        keep updated the internal state of the contract
74  def retriveNewBlock(bridgeContract):
75
76      blockNumber = 0
77      event_filter = bridgeContract.events.NewBlockAdded.
            createFilter(fromBlock='latest')
78
79      while True:
80      #if there is a new block
81          if(blockNumber != w3.eth.block_number):
82
83              blockNumber = w3.eth.block_number
84              blockHeader =
85                  RLPEncodeBlockHeader(w3,blockNumber)
86
87              txt_hash = bridgeContract.functions.
88                  saveBlock(blockHeader).transact()
89              txn_receipt =
90                  web3.eth.wait_for_transaction_receipt(
                        txt_hash)
91              print(txn_receipt)
92
93              received = False
94              #wait for the NewBlockAdded event that
                    certificate that the block header is saved
95              while not received:
96                          entries = event_filter.
                                get_new_entries()
97                          if (len(entries) > 0):
98                              received = True
99                              print(entries)
100                         else:
101                             time.sleep(3)
102             time.sleep(5)
103
104 def main():
105
106     contractPath = 'Contracts/Bridge.sol'
107     #compile contract
108     contractInterface = compile_source_file(contractPath)
109
110     #ask for the address of the bridge contract
111     print('enter the contract address')
112     _contractAddress = input()
113
114     #reference to the contract
```

```
115     bridgeContract = web3.eth.contract(
116         address=_contractAddress,
117         abi=contractInterface["abi"])
118 
119     print(
120         'BlockSaver has started\nContract: {}\nAddress\t: {}'
                .format( contractPath, _contractAddress)
121         )
122 
123     retriveNewBlock(bridgeContract)
124 
125 if __name__ == "__main__":
126     main()
```

# Appendix D

# listener.py

```
1  from web3 import Web3,HTTPProvider
2  from solcx import compile_source
3  from web3._utils.encoding import pad_bytes
4  import asyncio
5
6  def compile_source_file(file_path):
7
8          with open(file_path, 'r') as f:
9              source = f.read()
10         #returns the [abi,bin] tuple for the first element of
               the dictionary, i.e. the data related to Bridge
11         return compile_source(source,
12                               output_values=["abi", "bin"],
13                               solc_version="0.6.0").
14                                 get('<stdin>:Bridge')
15
16 # C2 is the Ethereum mainnet to witch we ask to retrieve the
       desired variable
17 w3 = Web3(HTTPProvider('https://evocative-stylish-isle.
       discover.quiknode.pro/6
       ad754e3368653a2665e62db8659b9f179a3ae43/'))
18
19 # web3 is the Web3 reference to the chain C1 for take a
       reference to the bride contract
20 chainAddress = 'http://127.0.0.1:8545'
21 web3 = Web3(HTTPProvider(chainAddress))
22 web3.eth.defaultAccount = web3.eth.accounts[0]
23
24
25
26
27
28
```

```
29 #function that asks c2 for the proof related to the value
       searched by the request
30 def handle_event(bridgeContract,event):
31
32     print("hendle the request")
33     requestId = event['args']['requestId']
34     account = event['args']['account']
35     key = event['args']['key']
36     blockId = event['args']['blockId']
37
38
39     proof = w3.eth.get_proof(account,[key], blockId)
40
41     StateProof = [
42         proof.address,
43         proof.accountProof,
44         proof.storageHash,
45         pad_bytes(b'\x00', 32, proof.storageProof[0].key),
46         proof.storageProof[0].value,
47         proof.storageProof[0].proof
48     ]
49
50     #call the verify function of OnChainContract
51     txt_hash = bridgeContract.functions.verify(
52         requestId,
53         _stateProof = StateProof,
54         _blockId = blockId).transact()
55     txn_receipt =
56         web3.eth.wait_for_transaction_receipt(txt_hash)
57     print(txn_receipt)
58
59 async def log_loop(contract, event_filter, poll_interval):
60     #waiting for RequestLogged events
61     while True:
62         for e in event_filter.get_new_entries():
63             handle_event(contract, e)
64         await asyncio.sleep(poll_interval)
65
66 def main():
67
68     print('enter the contract address')
69     contractAddress = input()
70
71     contractPath = 'Contracts/Bridge.sol'
72     #compile contract
73     contractInterface = compile_source_file(contractPath)
74
75     #reference to the contract
76     bridgeContract =
```

```
77          web3.eth.contract(address=contractAddress,
78                            abi=contractInterface["abi"])
79
80      #event_filter for the RequestLogged event
81      event_filter = bridgeContract.events.RequestLogged.
            createFilter(fromBlock='latest')
82
83      print('Listener has started\nChain\t: {}\nContract: {}\
            nAddress\t: {}'
84          .format(chainAddress, contractPath, contractAddress))
85
86
87      loop = asyncio.new_event_loop()
88      asyncio.set_event_loop(loop)
89      try:
90          loop.run_until_complete(
91              asyncio.gather(
92                  log_loop(bridgeContract, event_filter, 2)))
93      finally:
94          loop.close()
95
96  if __name__ == "__main__":
97      main()
```

# Bibliography


[1] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized business review* (2008), p. 21260.

[2] Amritraj Singh et al. "Sidechain technologies in blockchain networks: An examination and state-of-the-art review". In: *J. Netw. Comput. Appl.* 149 (2020). DOI: 10.1016/j.jnca.2019.102471. URL: https://doi.org/10.1016/j.jnca.2019.102471.

[3] Babu Pillai, Kamanashis Biswas, and Vallipuram Muthukkumarasamy. "Cross-chain interoperability among blockchain-based systems using transactions". In: *Knowl. Eng. Rev.* 35 (2020), e23. DOI: 10.1017/S0269888920000314. URL: https://doi.org/10.1017/S0269888920000314.

[4] *Scaling Ethereum through rollups. [Online, accessed the 4th of February 2023].* URL: https://ethereum.org/en/developers/docs/scaling/.

[5] Vladimir Braverman. "Sliding Window Algorithms". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. New York, NY: Springer New York, 2016, pp. 2006–2011. ISBN: 978-1-4939-2864-4. DOI: 10.1007/978-1-4939-2864-4_797. URL: https://doi.org/10.1007/978-1-4939-2864-4_797.

[6] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[7] E Knuth Donald et al. "The art of computer programming". In: *Sorting and searching* 3 (1999), pp. 426–458.

[8] Kamil Jezek. "Ethereum Data Structures". In: *CoRR* abs/2108.05513 (2021). arXiv: 2108.05513. URL: https://arxiv.org/abs/2108.05513.

[9] Edward Fredkin. "Trie memory". In: *Commun. ACM* 3.9 (1960), pp. 490–499. DOI: 10.1145/367390.367400. URL: https://doi.org/10.1145/367390.367400.

[10] Donald R. Morrison. "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *J. ACM* 15.4 (1968), pp. 514–534. DOI: 10.1145/321479.321481. URL: https://doi.org/10.1145/321479.321481.

[11] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32. URL: https://doi.org/10.1007/3-540-48184-2%5C_32.

[12] Guido Bertoni et al. "Keccak". In: *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Springer, 2013, pp. 313–314. DOI: 10.1007/978-3-642-38348-9_19. URL: https://doi.org/10.1007/978-3-642-38348-9%5C_19.

[13] *Ethereum. ethereum/wiki. [Online, accessed the 4th of February 2023]*. URL: https://github.com/ethereum/wiki/wiki/Ethash.

[14] David E. Fu and Jerome A. Solinas. "IKE and IKEv2 Authentication Using the Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *RFC* 4754 (2007), pp. 1–15. DOI: 10.17487/RFC4754. URL: https://doi.org/10.17487/RFC4754.

[15] *Solidity. Solidity language. [Online, accessed the 4th of February 2023]*. URL: https://docs.soliditylang.org/en/v0.8.17/.

[16] *Howard. How to decipher a smart contract method call [Online, accessed the 4th of February 2023]*. URL: https://medium.com/@hayeah/how-to-decipher-a-smart-contract-method-call-8ee980311603.

[17] *Solidity. Contract ABI specification. [Online, accessed the 4th of February 2023]*. URL: https://docs.soliditylang.org/en/v0.6.2/abi-spec.html.

[18] *Oracle. Designing a remote interface. [Online, accessed the 4th of February 2023]*. URL: https://docs.oracle.com/javase/tutorial/rmi/designing.html.

[19] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (1970), pp. 422–426. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[20] *Solidity. Storage Layout. [Online, accessed the 4th of February 2023].* URL: https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html.

[21] *Jean Cvllr. All About Solidity Data Locations — Storage [Online, accessed the 4th of February 2023].* URL: https://betterprogramming.pub/all-about-solidity-data-locations-part-i-storage-e50604bfc1ad.

[22] Seth Gilbert and Nancy A. Lynch. "Perspectives on the CAP Theorem". In: *Computer* 45.2 (2012), pp. 30–36. DOI: 10.1109/MC.2011.389. URL: https://doi.org/10.1109/MC.2011.389.

[23] *Lightning network. [Online, accessed the 4th of February 2023].* URL: https://lightning.network/.

[24] *Bitcoin. [Online, accessed the 4th of February 2023].* URL: https://bitcoin.org/it/.

[25] *How Do Payment Channels Work? [Online, accessed the 4th of February 2023].* URL: https://simplelightning.com/how-payment-channels-work-the-basics.html.

[26] *An introduction to the various types of cross-chain bridge solutions [Online, accessed the 4th of February 2023].* URL: https://cointelegraph.com/press-releases/an-introduction-to-the-various-types-of-cross-chain-bridge-solutions.

[27] *Cosmos whitepaper. [Online, accessed the 4th of February 2023].* URL: https://v1.cosmos.network/resources/whitepaper.

[28] *What is Tendermint. [Online, accessed the 4th of February 2023].* URL: https://docs.tendermint.com/v0.34/introduction/what-is-tendermint.html.

[29] Gavin Wood. "Polkadot: Vision for a heterogeneous multi-chain framework". In: *White paper* 21.2327 (2016), p. 4662.

[30] *Ethereum. Proof Of Stack. [Online, accessed the 4th of February 2023].* URL: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[31] *Analysis of Polkadot: Architecture, Internals, and Contradictions [Online, accessed the 4th of February 2023].* URL: https://www.smartcontractresearch.org/t/research-summary-analysis-of-polkadot-architecture-internals-and-contradictions/1960.

[32] Amritraj Singh. "PUBLIC BLOCKCHAIN SCALABILITY: ADVANCEMENTS, CHALLENGES AND THE FUTURE". In: (Apr. 2019).

[33] *Polygon Lightpaper. [Online, accessed the 4th of February 2023].* URL: https://polygon.technology/lightpaper-polygon.pdf.

[34] *Chainbridge Documentatio. [Online, accessed the 4th of February 2023].* URL: https://wiki.polygon.technology/docs/category/chainbridge/.

[35] *Harry A. Kalodner et al. "Arbitrum: Scalable, private smart contracts". [Online, accessed the 4th of February 2023].* URL: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner.

[36] *ZK-Rollups. [Online, accessed the 4th of February 2023].* URL: https://ethereum.org/en/developers/docs/scaling/zk-rollups/.

[37] Jacob Eberhardt and Stefan Tai. "ZoKrates - Scalable Privacy-Preserving Off-Chain Computations". In: *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*. IEEE, 2018, pp. 1084–1091. DOI: 10.1109/Cybermatics_2018.2018.00199. URL: https://doi.org/10.1109/Cybermatics%5C_2018.2018.00199.

[38] *Python. Python programming language [Online, accessed the 4th of February 2023].* URL: https://www.python.org.

[39] *Web3 library [Online, accessed the 4th of February 2023].* URL: https://web3py.readthedocs.io/en/v5/.

[40] *Ethereum. Gas and fees. [Online, accessed the 4th of February 2023].* URL: https://ethereum.org/en/developers/docs/gas/.

[41] *Ganache. [Online, accessed the 4th of February 2023].* URL: https://trufflesuite.com/ganache/.

[42] *QuickNode. [Online, accessed the 4th of February 2023].* URL: https://www.quicknode.com/.

[43] Tiancheng Xie et al. "zkBridge: Trustless Cross-chain Bridges Made Practical". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin et al. ACM, 2022, pp. 3003–3017. DOI: 10.1145/3548606.3560652. URL: https://doi.org/10.1145/3548606.3560652.

[44] Ilker Özçelik et al. "An Overview of Cryptographic Accumulators". In: *Proceedings of the 7th International Conference on Information Systems Security and Privacy, ICISSP 2021, Online Streaming, February 11-13, 2021*. Ed. by Paolo Mori, Gabriele Lenzini, and Steven Furnell. SCITEPRESS, 2021, pp. 661–669. DOI: 10.5220/0010337806610669. URL: https://doi.org/10.5220/0010337806610669.