École polytechnique fédérale de Lausanne

Sender Agnostic Batches for Carbon

Semester Project Report

Author: **Simone Roznowicz**

Responsible: **Prof. Rachid Guerraoui**

Supervisor: **Manuel Vidigueira**

10th June 2022

Contents

1 Introduction **2**
1.1 The game 2
1.2 Context 2
1.3 Motivation 3
1.4 Goals 3

2 Methods and Implementation **4**
2.1 Merkle Tree (MT) 4
2.2 Merkle Patricia Tree (MPT) 4
2.2.1 The purpose 4
2.2.2 How it is built 4
2.2.3 Siblings Retrieval and prove method 6
2.2.4 The minimum amount of information needed 7

3 Discussion **8**

4 Conclusion **9**

Chapter 1

Introduction

1.1 The game

In the development of a cryptocurrency like Carbon, the authentication process performs a pivotal role to guarantee secure and safe transactions. For the purpose of this project, Carbon's authentication process is described as a game in which three entities are present (see Figure 1.1):

- The **client** creates new transactions to perform and sends them to the broker.
- The **broker** is an intermediary between the client and the server: it has to transmit to the server the operations collected from the clients, after forming a *batch of transactions*. The broker also provides the server with a certificate that all the operations are correct: this is a cryptographic information (the aggregated multisignature, explained in the next paragraph), assumed to be small. The presence of the brokers is due to Carbon's attempt to accelerate the authentication and execution process of the cryptocurrency.
- The **server**, assumed to be correct, takes the two pieces of information given by the broker and either accepts or rejects the batch. More precisely, it validates the operations in the batch if and only if all of them were signed by the corresponding source clients. In other words, if each operation in the batch was approved by the corresponding source client, then the server validates the transactions.

This authentication game is proven to be correct even if any of the client or the broker is byzantine. In the game, the clients can only interface with the broker while the broker communicates with the clients and the server. Finally, the server interacts just with the broker.

1.2 Context

The mentioned authentication process can be summarized in the following steps:

1. The broker creates a batch of transactions to perform;
2. The broker builds a Merkle Tree (or a Merkle Patricia Tree as proposed afterwards) MT to store the batch of transactions;
3. For each source client in the batch, the broker creates a proof;
    (a) It sends a specific proof to each client,
    (b) Each client verifies the proof against the root considering its transactions,

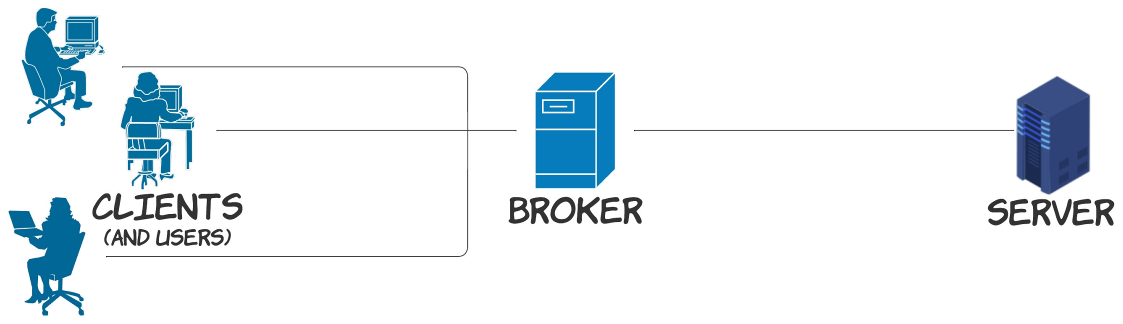

FIGURE 1.1

The picture shows the three mentioned entities, with the broker acting as the intermediary between the clients and the server.

   (c) The client multisigns the root and sends it back to broker,

4. The broker aggregates all the multisignatures into a single signature;
5. The broker sends the vector of items, together with the aggregated multisignature, to the validator (i.e. the server);
6. The validator reconstructs the MT: it creates *Valid-MT* on the vector of items;
7. The validator checks the multisignature against the *Valid-MT* hash of the root.

1.3 MOTIVATION

Currently, Carbon's signature verification relies on the mentioned data structure, named Merkle Tree, which permits to prove the inclusion of a transaction but not the exclusion: in other words, every client can perform at most one transaction per batch. Otherwise, the client can not verify that no other malicious transactions were inserted in the batch.

This is clearly a significative limitation to an extensive use of the cryptocurrency. Nevertheless, a specific data structure can avoid this problem: the Merkle Patricia Tree.

1.4 GOALS

The game focuses on efficiently dealing with step_2 and step_3. More precisely, the purpose is to present a specific implementation of a Merkle Patricia Tree such that the following two goals are reached:

1. Prove both inclusion and exclusion of a transaction in the batch: i.e. allow multiple transactions from the same client source in the batch (therefore making it agnostic to the client source).
2. Allow the brokers to send proofs to the clients, such that the minimum amount of information is efficiently sent over the network.

The following chapter will show more in details how these two objectives were reached.

Chapter 2

Methods and Implementation

2.1 Merkle Tree (MT)

Standard Merkle trees, as implemented in Carbon, can be built upon layers of vectors: therefore, they are generally balanced and require a minimum amount of storage memory. The order of key-value pairs in the tree is given by the insertion order; therefore, this MT does not require the presence of empty nodes. Standard MTs support only *Proof-of-Inclusion*, meaning that it is possible to verify whether a transaction is performed but it is not possible to certify that no other transactions with the same source client were introduced in the batch.

2.2 Merkle Patricia Tree (MPT)

2.2.1 The purpose

The First_Goal as already mentioned, is solved using a MPT which supports both *Proof-of-Inclusion* and *Proof-of-Exclusion*. The Second_Goal is solved sending over the network just the strictly necessary information, so that the client can compute the MPT hash of the root and multisign it.

2.2.2 How it is built

The MPT, in this project, consists of a binary tree in which every internal node presents two smart pointers, one to the left child and one to the right child. The leaves of the tree contain key-value pairs. Every node in the tree is associated with a hash, which is computed according to Cargo's_project. A picture of a MPT is represented in Figure 2.1

The modular MPT is implemented using Rust_programming_language and applying Generic Data Type whenever possible. The MPT is built upon two levels of abstractions: the first consists in the implementation of a generic node and three more specific ones, the second relies on the precedent implementation to effectively build the MPT and its methods. Here is the link for the project repository: ***https://github.com/SimoneRoznowicz/Semester-Project***.
A more detailed explanation of the two levels is presented below:

1. `NodeGeneric`: *enum* which can match one of the three following types:
    - `Internal`: every node having exactly two children. It presents three fields:
        - *left*: a smart pointer to the left child,

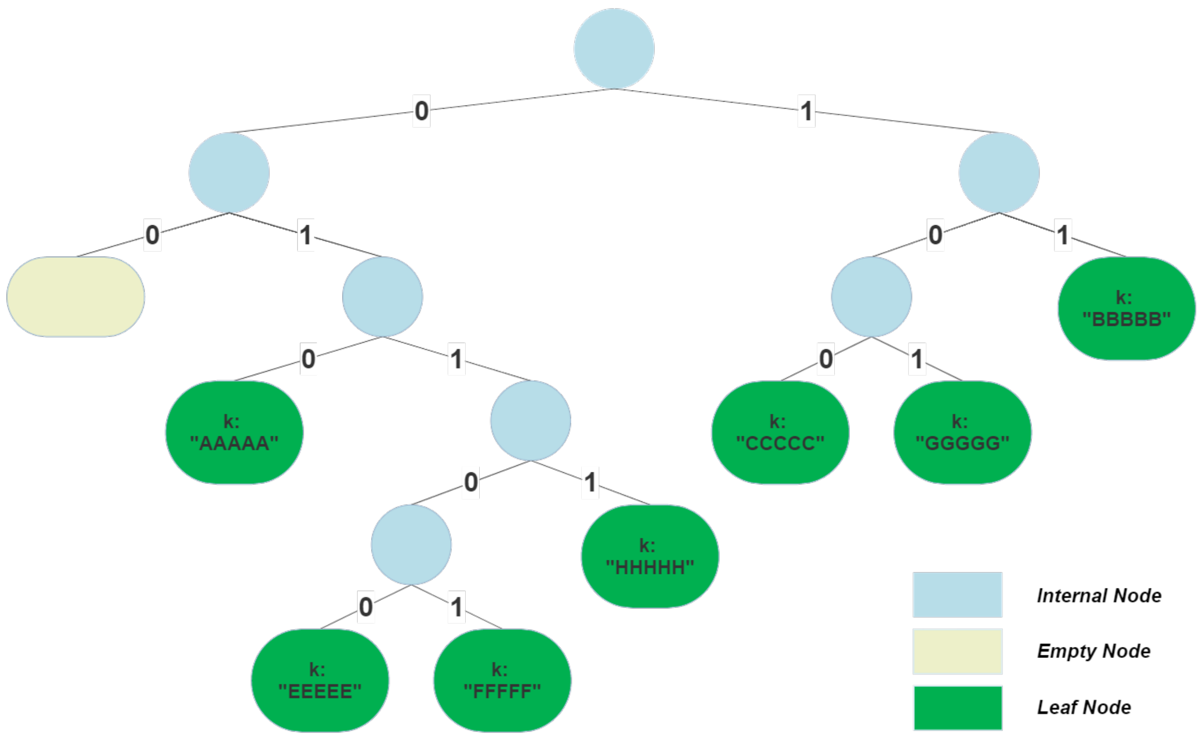

FIGURE 2.1

The picture shows a Merkle Patricia Tree: every Internal node has exactly two children; Leaf nodes are characterised by a key-value pair; finally, Empty nodes act as placeholders. NodeGeneric matches one of the these three types.

    - *right*: a smart pointer to the right child,
    - *my_hash*: a value which either represent the current hash of the node or is None.

- `Leaf`: every node containing the information of some transactions to be performed. It presents two fields:
    - *k*: the key to retrieve the node,
    - *v*: the value representing the list of the transactions having the same source client,
    - *my_hash*: a value which represent the hash of the node (this hash only depends on the two previous fields).
- `Empty`: every leaf node which does not contain any relevant information. It is still fundamental in order to keep the tree binary and to compute the hash of the internal nodes. It does not contain a field my_hash as this information would be redundant (every empty node has the same predetermined hash).

2. `MerkleTree`: a MPT whose purpose is storing the information of the batch and efficiently producing proofs for every specific client, when needed.

The research of the position where to insert a leaf or the retrieval of a node, as required in $insert$ and get_node methods respectively, is done according to a function named get_bit_direction: it takes as input an array of 32 unsigned integers (256 total bits), represented by the hash of a specific key, and outputs *true* if the bit at position *index* is 1, *false* otherwise.

The MPT presents some fundamental methods:

- *insert*: a recursive function which inserts a new Leaf in the MerkleTree if the key is not contained, otherwise it substitutes the current value of in the already existing leaf with the new one. While following the path in the binary tree according to get_bit_direction (using the $hash(key)$ as the 256 input bit array), to make the insertion efficient, the leaf is created at the minimum depth such that there is no ambiguity in retrieving the corresponding key. See Figure 2.2 and Figure 2.3
- *get_node*: a recursive function which returns a Result type: it contains a reference to the node associated to the given key, if the key is contained; *Err(())* otherwise.
- *compute_hashes*: a recursive function which computes the hash of every node applying a bottom-up approach (from the leaves to the root). Assigns the corresponding hash to *my_hash* member variable. Eventually returns the hash of the root.

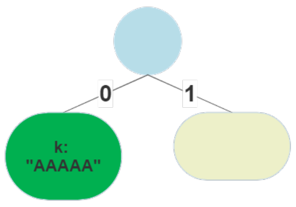

FIGURE 2.2

Initially, the MPT comprises one leaf whose key is the string *"AAAAA"*.

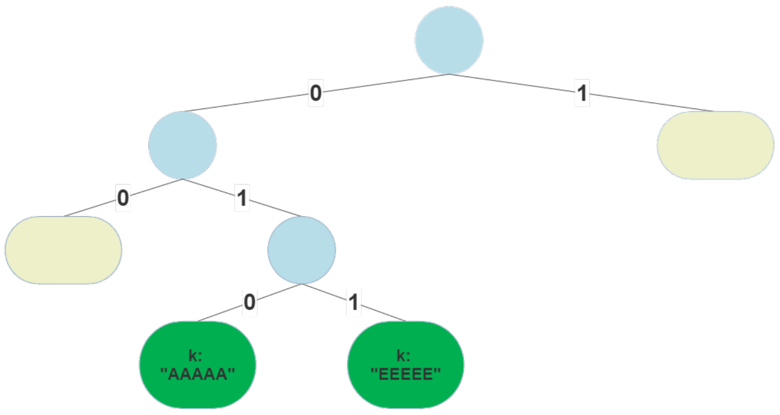

FIGURE 2.3

Now a new key-value pair has to be inserted. considering the hash of a key as just a string of 256 bits, the first 8 bits for the two keys are the following:
*hash("AAAAA") = 01**000100**...*
*hash("EEEEE") = 01**100100**...*
The first two bits are the same. Therefore, to solve the ambiguity, it is necessary to push the leaves two steps more into depth. As the tree must remain binary, empty nodes are created when required, as visible in the picture.

2.2.3 SIBLINGS RETRIEVAL AND PROVE METHOD

The process of producing a proof is performed by the broker. A proof contains a vector of siblings. Each sibling consists of two entries, a *hash* and a *direction*: the former is simply the computed value my_hash of a given sibling node, the latter is the position (left or right) of the sibling with respect to the other child. The actions of producing a proof and using it to compute the MPT hash of the root translate into two linked methods:

- *prove*: an operation performed by the broker. Given a specific client source, returns a proof containing a vector of siblings. Namely, the proof contains the strictly necessary information, i.e. other sibling nodes of the tree, so that the client is able to recompute the hash of the root, without rebuilding the entire MPT.
- *get_root_hash*: an operation performed by the client. Given a specific proof, it computes the hash of the root: the initial hash of the leaf (corresponding to the source client) is iteratively updated by

using the information contained in every sibling, as shown in Figure 2.4.

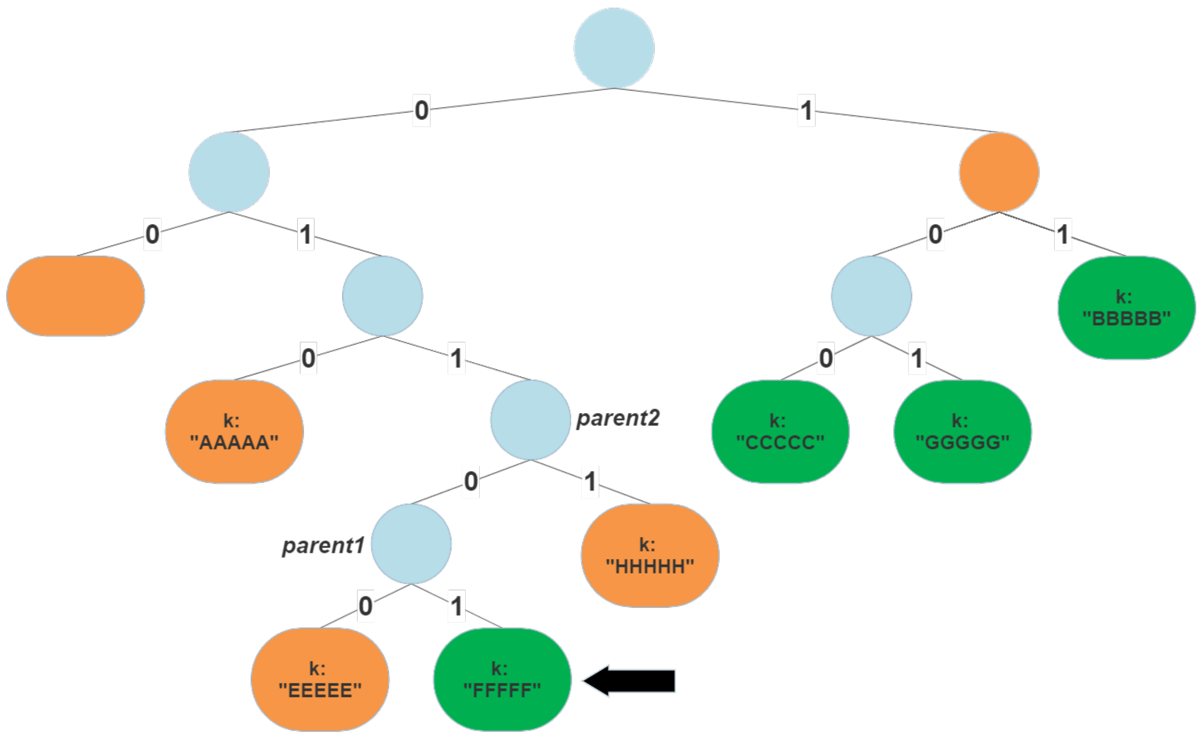

FIGURE 2.4

The figure presents a simple example of a MPT. Assuming the broker created a proof for the client having the string key *"FFFFF"*, then the orange nodes represent all the siblings contained in a vector in the proof. The get_root_hash method computes the hash of the leaf (remind that the client clearly knows its key and the transactions that it wants to perform), then computes the hash of the parent as following:
hash_parent1 = hash (hash(sibling1), hash(leaf))).
Similarly: hash_parent2 = hash (hash(parent1)), hash(sibling2)). Iterating the process for all the siblings, eventually, the hash of the root is calculated with the minimum amount of information.

2.2.4 THE MINIMUM AMOUNT OF INFORMATION NEEDED

The client does not receive all the batch of transactions: this could potentially flood the network and cause expensive computations to derive the MPT hash of the root. Instead, the purpose of the client, in the method get_root_hash, is to compute the MPT hash of the root starting from its key, a known vector of transactions and a vector of siblings, sent by the broker. Considering n total nodes of the MPT, the former expensive solution requires computing $O(n)$ hashes and creating $O(n)$ nodes. On the other side, the proposed alternative requires to just compute one hash per level of the tree, resulting in having a $O(\log n)$ time complexity. Moreover, as no additional node is created, the space complexity is $O(1)$.

CHAPTER 3

DISCUSSION

The implementation of the MPT, as described in the previous chapter, permits to create a proof for every client which is source of at least one transaction. Some extended tests were performed to ultimately attest the correctness of the previous reasoning. Eventually, it was indeed possible to create a test (get_root_hash_test) which does the following:

- creates a new MPT;
- inserts a certain number of leaves;
- defines hash_root as the hash of the root computed by the broker after creating the MPT;
- creates a proof for a specific client (i.e. for a specific key);
- defines reconstructed_hash_root as the hash of the root computed by the client as described in the method *get_root_hash*;
- correctly verifies that hash_root == reconstructed_hash_root;

However, an MPT is just one of the possible solutions and presents a relatively small downside. Indeed, the length of some edges can be really important compared to the others, translating into a worse retrieval performance of some leaf nodes. This is due to a possible situation in which at least two keys have the same n characters (where n is an integer such that $0 \leq n \leq 255$ but in this unlucky case it is big). A simple MT, as implemented in Carbon, instead, is more balanced, meaning that every node at the lowest level can be at depth d, $d + 1$ or $d - 1$ (where d is an integer such that $1 \leq d \leq 254$).

The use of an MPT presents this drawback with respect to the MT that is paid to ensure proof-of-exclusion.

CHAPTER 4

CONCLUSION

The First_Goal was solved by creating a designed MPT. The inclusion proof can be guaranteed in find_path method: a certain key, if present, will be retrieved together with the associated value. The exclusion proof is then assured by the value of a leaf, which is assumed to be a vector containing all the transactions that want to be performed by a specific source client. Concerning the Second_Goal, as already discussed, the choice of sending over the network just a vector of siblings as proof, has revealed to be an optimal solution permitting the client to efficiently compute the MPT hash of the root. In conclusion, despite a slightly worse performance while retrieving a key in some unlucky cases, the given implementation permitted to successfully reach both of the goals.