Blockchain interoperability: Merkle proofs

Plan

Abstract

This post is intended to present the minimal knowledge required for the understanding of how interoperability works and gives a quick overview of some current solutions.

We first present the background needed in blockchain technology with a focus on Merkle trees as they are mainly used in interoperability solutions.

We then define interoperability and explain how some interesting protocols are achieving it.

Background

For two blockchains to be interoperable, they must understand what happened in them, they must understand their data structure. Since [7], merkle trees are widely used in blockchain. We introduce merkle trees and inclusion proofs and how they are used in blockchain. You can also follow the notebook [3] in order to better understand this chapter.

1.1 Merkle tree [4]

Ralph Merkle invented the Merkle trees in 1979. We present here the binary Merkle tree used in Bitcoin. We present also a derivation of the Patricia trie used in ethereum in 1.1.7.

1.1.1 Binary Merkle tree

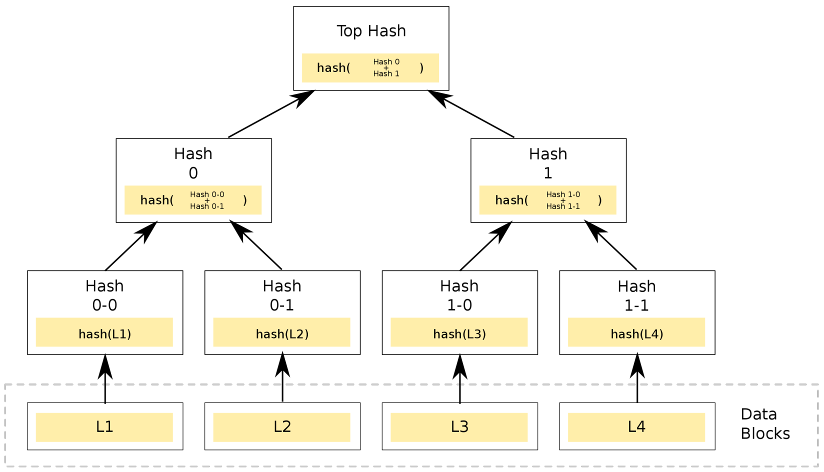


Figure 1.1: Binary Merkle tree from [6]

A binary Merkle tree is the result of hashing blockchain transaction two by two:

1. List all the data that need to be saved in an ordered list: it can be n user ids, for example, or any kind of data. In blockchain, these data are the transactions. Here say we have 4 transactions L1, L2, L3, and L4.

2. Hash these data one by one to obtain n hashes. Here we obtain 4 transactions hashes $(Hash_{0-0}, Hash_{0-1}, Hash_{1-0}, Hash_{1-1})$

3. Hash the hashes two by two starting from the left. If there is only one node on the right of the tree, we duplicate it to obtain the parent. Here we obtain $Hash_0$ and $Hash_1$.

4. Repeat the last step until we obtain only one node left: the root.

The root is a fingerprint of all the data, in our case the four transactions. It means that if one changes anything in the original data, the timestamp of the first transaction, for example, it'll change completely the root hash of the Merkle tree, thanks to the non-locality property of the hash function used (Alice and Aline have very different hashes even if they have only one different letter). The security of Merkle trees relies on the collision resistance of their hash function.

To better understand this process, we'll take the example of a bitcoin Merkle tree for the block 170861:

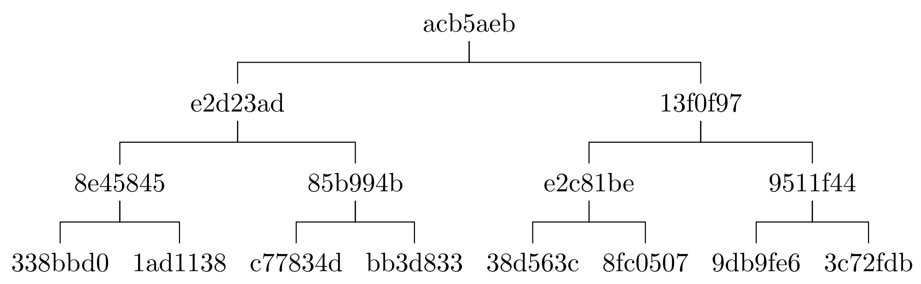


Figure 1.2: Merkle tree of the bitcoin block 170861

1.1.2 Proof of inclusion

Merkle trees allow to summarize the data into one fixed-length fingerprint, the root hash. But why should we use this data structure instead of just hashing all the hashes for example?

This data structure allows proof of inclusion. In the case of bitcoin for example, the data is the transactions, and this structure allows other users to answer the folloxing question: *Is this transaction really included in that block?*

So let's consider the Merkle tree below from our example:

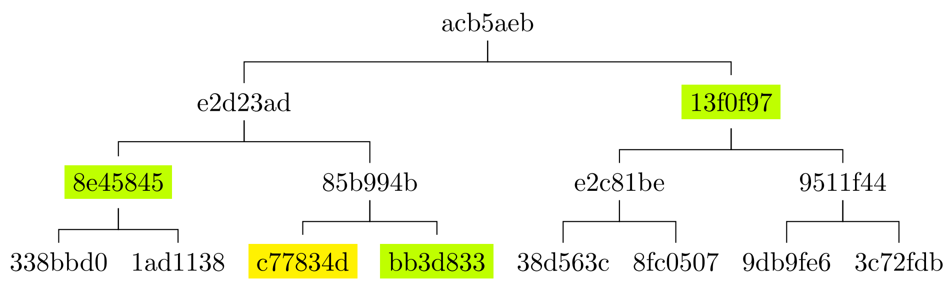


Figure 1.3: Proof of inclusion: Transaction to be verified in yellow, Merkle proof in green

We have 8 transaction hashes that we want to place in the Merkle tree in this example. Let's say we are interested in the transaction in yellow (c77834d) because it's directer towards to our account. We want a proof that this transaction has really happened, and has been included in the block with Merkle root acb5aeb.

The proof is a Merkle block with the green hashes. The verifier of this proof can then follow these steps.

1. With the yellow transaction and the green one bb3d833 you can concatenate and hash it to get the parent hash 85b994b.

2. With this last hash 85b994b and the green one 8e45845 you can concatenate and hash them to get the parent hash e2d23ad.
3. And with this last hash concatened with the last green one 13f0f97 and hashed, we get the Merkle root
4. We compare the calculated Merkle root from the proof and the real one from the block. If they match, the proof is accepted. In Bitcoin, it means that the transaction is included in this block. If it doesn't, we cannot prove the inclusion with that proof.

1.1.3 Proof scaling

What is the advantage of using this data structure?

The verifier only needs the Merkle root of the block to accept the proof of inclusion. But as this is a binary tree, the number of hashes at each level is divided by two, so this proof only consists of $\log_2(n)$ hashes, with n the number of transactions. It means that if we have 8 transactions, as in this example we need one hash per level so $\log_2(8) = 3$ hashes to prove the inclusion of the transaction in the block.

If $n = 100$ *we need* $\log_2(100) \sim 7$ *hashes; if* $n = 1000$ *we need* $\log_2(1000) \sim 10$ *hashes.*

Now there is a trade off. You can reduce the height of the tree by making it wider. Instead of two children and the binary tree from Bitcoin, we can have 16 children (plus the value) like in Ethereum. This has a drawback as it increases the length of the proof.

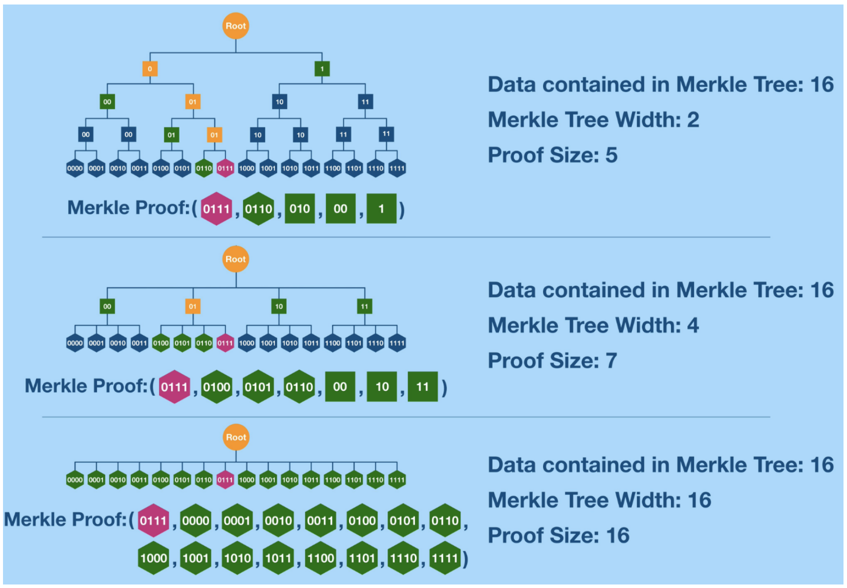

Root
Data contained in Merkle Tree: 16
Merkle Tree Width: 2
Proof Size: 5
Merkle Proof:(0111 , 0110 , 010 , 00 , 1)
Data contained in Merkle Tree: 16
Merkle Tree Width: 4
Proof Size: 7
Merkle Proof:(0111 , 0100 , 0101 , 0110 , 00 , 10 , 11)
Data contained in Merkle Tree: 16
Merkle Tree Width: 16
Proof Size: 16
Merkle Proof:(0111 , 0000 , 0001 , 0010 , 0011 , 0100 , 0101 , 0110 , 1000 , 1001 , 1010 , 1011 , 1100 , 1101 , 1110 , 1111)

Figure 1.4: Merkle tree proof scaling [5]

The general formula for calculating the proof's length is $\text{length}_{proof} = (m-1)\log_m(n)$ with n leaves of width m.

Note that Verkle trees [13] are an interesting solution to this problem.

1.1.4 Some applications

Merkle trees play a key role in distributed systems because the same data should exist in multiple places. Here the data integrity property is important.

Blockchains, since the appearance of their white paper, use the Merkle trees to represent data. This allows data integrity as for any distributed systems but also allows the light nodes to just download the fingerprint of the data.

Sismo [9] manages groups of accounts represented in Merkle trees. Accounts are a key value pair: the key is the identifier, usually the address of the user, and the value depends on the group. The groups are for example Twitter accounts, where the value is the number of followers, or github contributors, where the value is the number of commits.

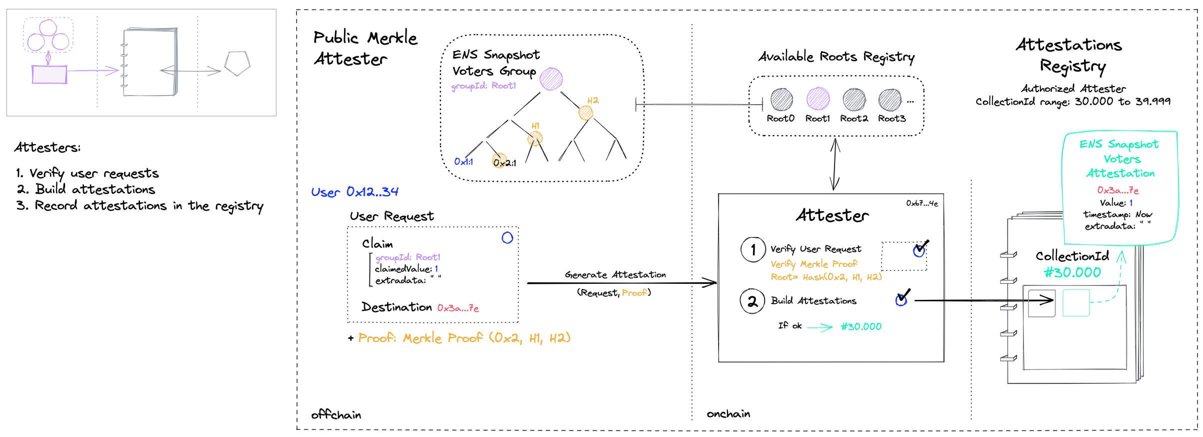


Figure 1.5: Sismo: users claim their attestation.[9]

1.1.5 Ethereum tree

Binary Merkle tree are used in Bitcoin. In Ethereum a more complicated version is used, derived from the Patricia trie. First let's understand what a trie is.

1.1.5.1 Trie

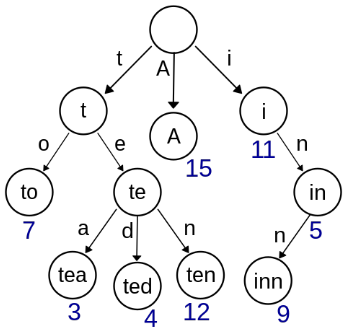


Figure 1.6: Trie representing the words ”A”, ”to”, ”tea”, ”ted”, ”ten”, ”i”, ”in”, and ”inn”

A trie is a tree data structure where the key of the node is also his path from the root. The key is usually a string. In other words positions in the trie are determined directly from the key and the inverse is correct.

Let's say you want to save n words in the trie.

1. From the root, at step 1 we take the first letter of each of the n words and create a leaf for each different ones. In the example from the figure above, we take A from A, t from to, tea, ted, ten, i from i, in and inn. We have then 3 nodes from the root: t, A, and i.

2. At step t, repeat the first step but instead take the t first letters.

3. Stop when all the original words are in a leaf.

If we have a string with 200 letters, we'll have to create 200 levels.

1.1.6 Patricia trie

As said in the last section, if you have a n-length word you'll have to create n levels, which is not optimized in term of storage. Instead, radix trees and Patricia tries are a compressed version of the above trie.

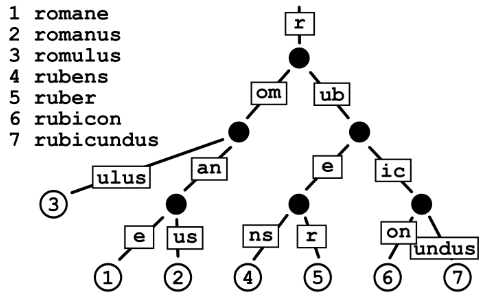


Figure 1.7: Patricia trie

Here, we are not limited at each step in terms of length of the key.

1.1.7 Patricia Merkle trie

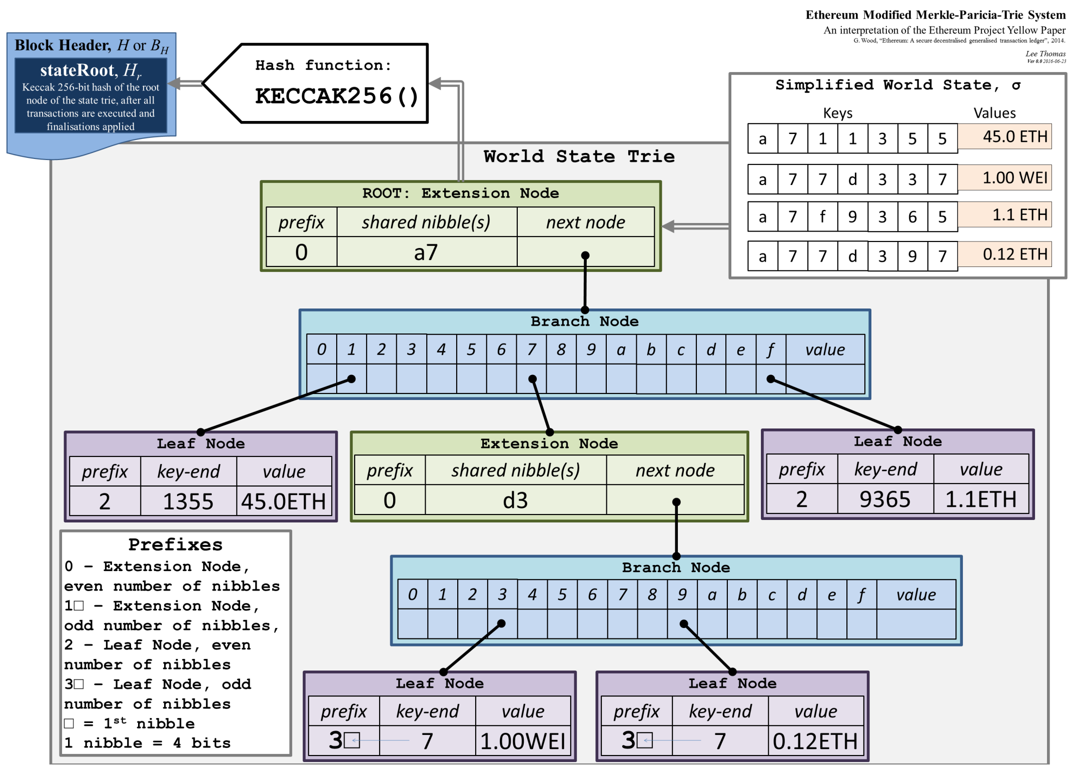

Ethereum Modified Merkle-Paricia-Trie System
An interpretation of the Ethereum Project Yellow Paper
G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", 2014.
Lee Thomas
Block Header, H or B_H
stateRoot, H_r
Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied
Hash function:
KECCAK256()
Simplified World State, σ
Keys
Values
a 7 1 1 3 5 5 45.0 ETH
a 7 7 d 3 3 7 1.00 WEI
a 7 f 9 3 6 5 1.1 ETH
a 7 7 d 3 9 7 0.12 ETH
World State Trie
ROOT: Extension Node
prefix
shared nibble(s)
next node
0
a7
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
Leaf Node
prefix
key-end
value
2
1355
45.0ETH
Extension Node
prefix
shared nibble(s)
next node
0
d3
Leaf Node
prefix
key-end
value
2
9365
1.1ETH
Prefixes
0 – Extension Node, even number of nibbles
1□ – Extension Node, odd number of nibbles,
2 – Leaf Node, even number of nibbles
3□ – Leaf Node, odd number of nibbles
□ = 1st nibble
1 nibble = 4 bits
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
Leaf Node
prefix
key-end
value
3□
7
1.00WEI
Leaf Node
prefix
key-end
value
3□
7
0.12ETH

Figure 1.8: Ethereum Patricia trie

The Patricia Merkle trie used in Ethereum modifies the original Patricia trie. The keys are hexadecimal strings and defines 4 types of nodes:

- EmptyNode, represented as the empty string;
- BranchNode, a 17-item node [v0 ... v15, vt];
- LeafNode, a 2-item node [encodedPath, value];
- ExtensionNode, a 2-item node [encodedPath, key].

A branch node at a certain step t, at letter t, is created when there are multiple possible letters for the next letter at $t + 1$. Since we use hexadecimal keys, at each step we can choose between 16 letters (or the value).

A leaf node is the last possible node in the branch and contains the value. Here we can say that the key represents the address of an Ethereum account, and the value is the balance.

The extension node is the compression offered by the Patricia trie. It represents the shared nibbles (letters) before two or more words diverge.

Ethereum [1] stores three Merkle roots in the block header: one for the transactions, as in Bitcoin, another one for the state (balances for example), and another one for the receipts.

1.2 Zero knowledge

This section is meant to be a light introduction to the zero-knowledge-proof concept to understand interoperability solutions.

A zero-knowledge protocol is a method by which one party (the prover) can prove to another party (the verifier) that something is true, without revealing any information apart from the fact that this specific statement is true [2].

For example, in the case of digital signatures, you send a signature proving that you are indeed the one sending the mail. More precisely, this signature proves that you know the private key without revealing it. The verifier checks that the signature matches the public key of the sender ($\sim$ the signature reveals the public key)

Instead of only proving that you know the private key as in digital signatures, zero-knowledge proofs can prove "any type" of computation. For example, you can prove that:

- you know m such that $SHA_3(m) = 0$ to the verifier, without revealing m;
- you know a sudoku solution to a sudoku problem without revealing it;
- you know the 1000^{th} Fibonacci number (defined by the suite $a_{n-2} + a_{n-1} = a_n$);
- some transactions are valid, thus allowing for instance ZK roll-ups to process transactions off chain, and prove on-chain to Layer 1 that these transactions are indeed valid.

In fact, zero-knowledge (ZK) proofs can offer computational integrity and privacy (with zero knowledge). Privacy is not always necessary and ZK can be used only for the computational integrity property.

Computational integrity proves that you have done the right computation, even when no one is watching. For example, in ZK rollups, computational integrity is the main need, and the privacy is not always/often taken into account.

Privacy with zero knowledge ensures thaty you don't reveal anything about the statement. In Zerocash [8], privacy is needed to ensure the confidentiality of the transactions.

A zero-knowledge protocol should verify the following properties.

Completeness If the claim is valid, the verifier accepts the proof.

Soundness If the claim is invalid, the verifier should reject the proof with very high probability.

Zero-knowledge The verifier learns nothing about a statement beyond its validity or falsity.

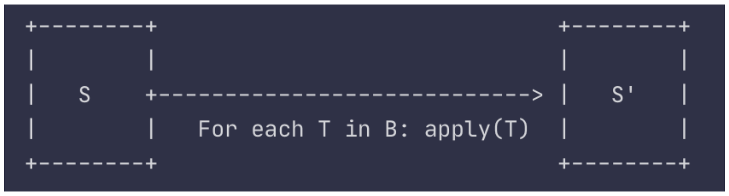
To conclude, some important properties are expected from these protocols. We want short proofs in terms of size, for example. We also want, and this is a fundamental requirement, very fast verifications. If the prover with a highly efficient computer computes the hash of a 200GB document, the verification should be very fast and consists in a much lighter computation than the prover's one.

This last requirement is at the heart of the zero-knowledge research and explains the complexity of the techniques used.

1.3 Blockchain basics

A blockchain can be viewed as a replicated state machine. It means that each time t is associated with a global state that goes to the next state at $t + 1$ with the state transition function.

Here the state is for example account balances for Bitcoin, and the state transition function represented by the transactions, instructions to be executed to go to the next state. In blockchains, the transactions for each state transition are placed in a new block, adding to the chain of blocks.



```
+--------+                            +--------+
|        |                            |        |
|   S    +--------------------------> |   S'   |
|        |   For each T in B: apply(T) |        |
+--------+                            +--------+
```

Figure 1.9: The state S transitioning to state S' by executing all transactions T from the block B [10]

Lastly, it is replicated across all the full nodes to ensure global consistency of the distributed ledger, meaning that the nodes on the network should agree on the blockchain state thanks to the consensus algorithm.

1.3.1 The block

So what is stored exactly in a block?

Let's say for the moment that a Bitcoin block consists of:

- the hash of the previous block (you take the whole previous block and pass it to the hash function);
- the nonce resulting from the cryptographic puzzle (we'll talk about it later);
- the list of transactions included in the block.

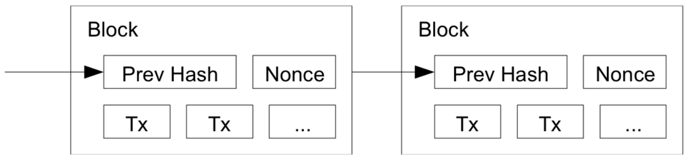


Figure 1.10: A Bitcoin block: the hash of the previous block, the nonce, and the list of transactions [7]

This requires every node in the network to keep the list of transactions since 2009.

1.3.2 Merkle tree of transactions

In fact, the block is composed of the block header and the list of transactions. The block header contains this time the root hash of the transactions Merkle tree. The root hash is a summary of all the transactions happening during that period represented a Merkle tree. For Bitcoin, binary Merkle tree are used for transactions, whereas, in Ethereum, Merkle Patricia trees are used to store transactions, state, and receipts.

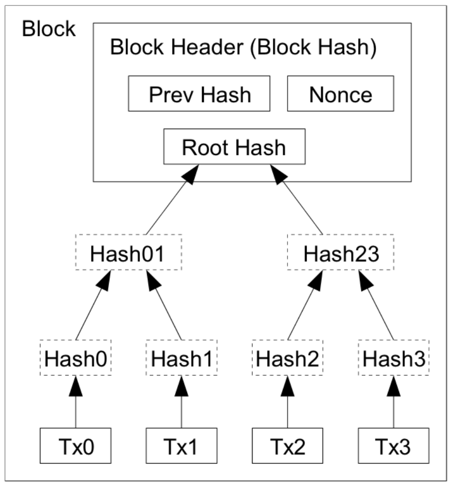


Figure 1.11: A Bitcoin [7] block

1.3.3 Single payment verification

In [7], the author mentioned merkle inclusion proofs to verify payment, but in order to keep only the block headers in the blockchain. For security reasons this never happened.

Instead this technique allowed two actors to emerge.

A **full node** stores all the full blocks, with the transactions. With these transactions, it can provide the proof of inclusion: the hashes required to prove a transaction is in a particular block.

A **light node** asks for the blocks headers to the full nodes of the network. Once it got the block headers, it can ask the full nodes of the network to provide a proof for the transactions it is interested in. With the procedure described in Section 1.1.2, the light node can verify that a specific transaction has indeed happened in the blockchain.

A light node is only storing the block headers, so even a phone can handle the amount of storage needed. This allows for example to verify with your wallet that a transaction has been executed on your laptop without downloading the $\sim$ 500GB of the blockchain.

1.3.4 Security of light clients

We now share some security considerations about the block headers validity/finality. The light clients are able to verify the proofs of inclusion against the block header Merkle root. But how can they trust this block header? Is it final ? Is it valid?

1.3.4.1 Proof of work

In Bitcoin, with the proof-of-work mechanism, light clients ask the full nodes for the last block headers and select the longest chain. A block in Bitcoin and with proof of work contains a nonce (number generated once). This nonce represents the amount of work done by the validator to submit this block. Indeed the miners, full nodes with the ability to produce blocks, try to solve a cryptographic puzzle. This puzzle consists in testing all values of the nonce until they find one that makes the hash of the block (including the nonce) starting with a certain number of zeros. In other words, they have to brute-force search a nonce such that the block hash meets the required level of difficulty: the number of zeros. For example, if we ask the miners to find a hash of the block with only one zero, it'll be pretty easy. If we ask them to find a block hash with 10 starting zero, it'll be a lot more difficult.

This nonce is the probabilistic guarantee that the block header is valid. The more blocks accepted (with a nonce then) after that block confirm that the transactions for this block and block header are valid and will not be reverted.

1.3.4.2 Proof of stake

If the proof-of-work mechanism relies on the amount of work done to ensure the security of the light client, the security of the proof-of-stake light client is a less obvious.

As a proof-of-stake light client, we can get the genesis header and sync to the latest block header from it. But it's slow and unsafe because it's costless for a hacker to fork the network at some point and produce invalid blocks (long-range attacks). Instead, a light client can ask a trusted source for header that is not in the unbounding window. It means that the trusted source, full node, can be slashed if it gives the light client fake block headers. The trusted source can be someone you know, a friend or a verified website for example.

We recall that the Ethereum 2.0 has a sync committee of 512 validators randomly chosen every 27 hours; it is responsible for signing every block header during that period. If at least 2/3 of the validators sign a given block header, the block is considered finalized and the network state is valid. This last consideration is important for the solutions presented in the next chapter.

Interoperability

Merkle proofs from Section 1.1.2 are used to prove to a light client that a transaction has happened. **Now this can also be used to prove to blockchain B that a transaction like a token transfer has taken place in a blockchain A.**

In this chapter let's see how interoperability between blockchains can be achieved using Merkle proofs.

2.1 Interoperability with Merkle proofs

2.1.1 Definition

Interoperability has many different definitions. For example, [14] defines it as the ability of two or more software components to cooperate despite differences in language, interface, and execution platform.

Now blockchain interoperability can be defined by the ability of different blockchain networks to communicate and exchange data with each other.

2.1.2 Token-transfer workflow

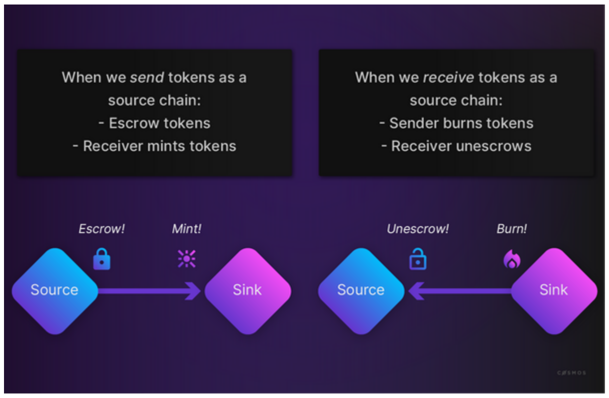


Figure 2.1: Token transfer [12]

1. The user from the source chain send his tokens to the receiver to an escrow contract deployed on the source chain.
2. Full nodes of chain A monitor these events (usually).
3. They then generate the Merkle proof for the transaction of these escrowed tokens.
4. The blockchain B receives, somehow, the proof that the transaction happened along with the block header of this transaction.
5. Blockchain B can then be sure that the transaction happened and mint the equivalent tokens to the receiver.

2.1.3 Expressivity

The question is: how well can you describe the source chain in order to trigger actions in the destination chain? In Bitcoin we just have the binary Merkle root of the transactions. It means that you can only prove the inclusion of the transactions. How many bitcoins do you have? It's not a question you can answer with the Merkle root in Bitcoin, because it only describes the transactions.

In Ethereum in contrast, as explained here 1.1.7, there are three trees saved: one for the transactions, as in Bitcoin, one for the state, and one for the receipts. **These Merkle trees allow Ethereum to answer more questions than just the inclusion of a transaction** , for example:

1. has this transaction been included in a particular block?
2. what is the current balance of my account?

3. does this account exist?

4. what are all instances of an event of type X (e.g., a crowdfunding contract reaching its goal) emitted by this address in the past 30 days ?

5. if you pretend to run this transaction on this contract, what would the output be?

The first one can be answered by the transactions tree as in Bitcoin, the second and the third by the state tree, and the fourth by the receipt tree. The fifth is a bit more complicated but can be proven by the state tree also.

Obviously Ethereum offers much more functionalities than Bitcoin, and these trees offer the ability to describe the Ethereum world more precisely.

This is relevant for the interoperability challenges. You can now react in the destination chain to a lot more information from the source chain.

2.2 Examples

So, now that we understand the Merkle proofs, **how are they used in current interoperability solutions?**

We present a rapid overview of some interesting solutions; the goal here is to understand where the Merkle proofs step in and how they help achieving interoperability in these protocols, leaving many details uncovered to keep it simple.

In Section 1.3.4, we highlight that the security of light clients relies on having the verified block headers. The following protocols use different techniques to prove that the block headers are valid.

2.2.1 LayerZero [16]

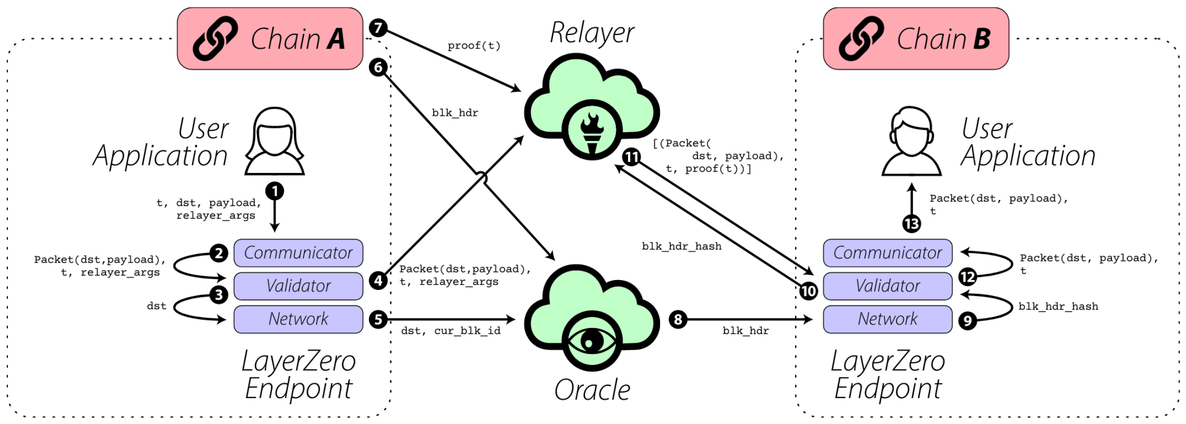


Figure 2.2: LayerZero [16]

For each supported chain, LayerZero deploys an endpoint, consisting in 4 smart contracts. This endpoint includes a library handling the verification of the transaction proof.

So, from Section 1.1.2, to verify that a transaction has taken place we need the Merkle root included in the block header and the Merkle Proof. LayerZero offers interoperability by transporting these two necessary pieces of data.

- The block header containing the Merkle root is carried by the oracle from blockchain A to blockchain B.
- The Merkle proof of blockchain A is carried by a relayer from blockchain A to blockchain B.

The oracle is here a Chainlink oracle monitoring the source blockchain, and the relayer can be handled by LayerZero or anyone interested into this.

Chain B can then attest that the transaction has happened by verifying the proof against the block header sent by the oracle.

The security of LayerZero relies on the independence of the relayer and the oracle. If the oracle and the relayer collude, they can give a fake block header and a proof working with that fake block.

2.2.2 Telepathy [11]

We present Telepathy and cite [15] which is similar in their approach to validate the block header using zero knowledge proof.

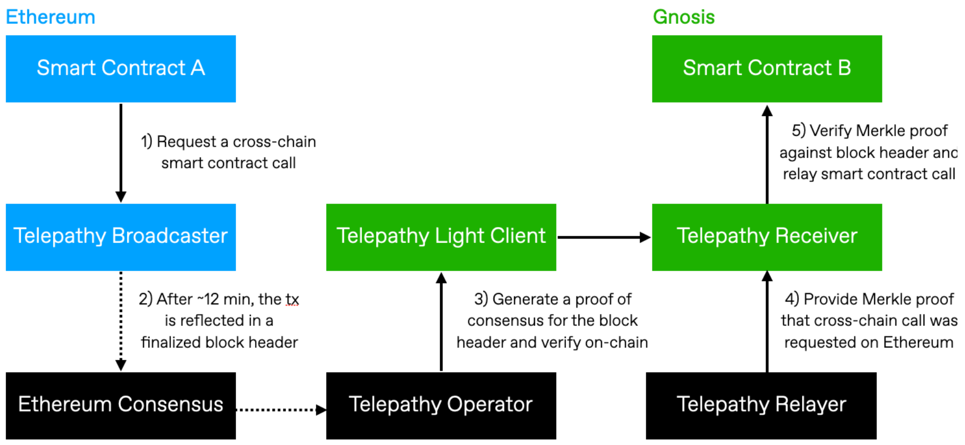

Ethereum
Gnosis
Smart Contract A
Smart Contract B
1) Request a cross-chain smart contract call
5) Verify Merkle proof against block header and relay smart contract call
Telepathy Broadcaster
Telepathy Light Client
Telepathy Receiver
2) After ~12 min, the tx is reflected in a finalized block header
3) Generate a proof of consensus for the block header and verify on-chain
4) Provide Merkle proof that cross-chain call was requested on Ethereum
Ethereum Consensus
Telepathy Operator
Telepathy Relayer

Figure 2.3: Telepathy workflow [11]

We recall from 1.3.4.2 that the Ethereum sync committee validators have to sign the block headers by a 2/3 majority.

Telepathy is based on a consensus proof. It means that to prove that a block header from blockchain A is valid, it will basically check that a large majority of the validators signed the block header.

But verifying all these signatures one by one is computationally expensive. Instead, the proof mechanism used in Telepathy is a zero-knowledge proof attesting that the block header has been signed by a sufficient percentage of validators.

The zero-knowledge proof can be cheaply verified onchain, and chain B can then expose these verified blockheaders. The smart contract in chain B can then verify the Merkle proof relayed by the Telepathy relayer against the verified block header.

The security of the Telepathy relies directly on the honesty of the Ethereum validator set (sync committee validators).

Références

[1] Vitalik Buterin. "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform". In: (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[2] Shafi Goldwasser, Silvio Micali, and Chales Rackoff. "The Knowledge Complexity of Interactive Proof-Systems". In: *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 203–225. ISBN: 9781450372664. URL: https://doi.org/10.1145/3335741.3335750.

[3] *Inclusion proofs notebook*. https://github.com/ggkkdev/post_interoperability/blob/main/merkle_bitcoin.ipynb.

[4] Ralph C Merkle. "A digital signature based on a conventional encryption function". In: *Advances in Cryptology—CRYPTO'87: Proceedings 7*. Springer. 1988, pp. 369–378.

[5] *Merkle proof scaling*. https://twitter.com/SalomonCrypto/status/1586391080727064577.

[6] *Merkle trees*. https://fr.wikipedia.org/wiki/Arbre_de_Merkle.

[7] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (May 2009). URL: http://www.bitcoin.org/bitcoin.pdf.

[8] Eli Ben Sasson et al. "Zerocash: Decentralized anonymous payments from bitcoin". In: *2014 IEEE symposium on security and privacy*. IEEE. 2014, pp. 459–474.

[9] *Sismo*. https://docs.sismo.io/sismo-docs/technical-documentation/zk-badges-protocol/groups.

[10] *State machine*. https://docs.cosmos.network/v0.46/intro/sdk-app-architecture.html.

[11] *Telepathy docs*. https://docs.telepathy.xyz/.

[12] *tokentransfer*. https://tutorials.cosmos.network/academy/3-ibc/5-token-transfer.html.

[13] *Verkle trees*. https://vitalik.ca/general/2021/06/18/verkle.html.

[14] Peter Wegner. "Interoperability". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 285–287.

[15] Tiancheng Xie et al. “zkbridge: Trustless cross-chain bridges made practical”. In: *arXiv preprint arXiv:2210.00264* (2022).

[16] Ryan Zarick, Bryan Pellegrino, and Caleb Banister. “Layerzero: Trustless omnichain interoperability protocol”. In: *arXiv preprint arXiv:2110.13871* (2021).