Blockchain interoperability: Merkle proofs

Plan

Abstract

This post is intended to present the minimal understanding of how interoperability works and give a quick overview of some current solutions. For example, the details of the merkle trees in Ethereum are left in the appendix as it adds complexity and the post is hopefully understandable without it.
We first present the background needed in blockchain 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

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 tree used in ethereum in A.1.3.

1.1.1 Binary merkle tree

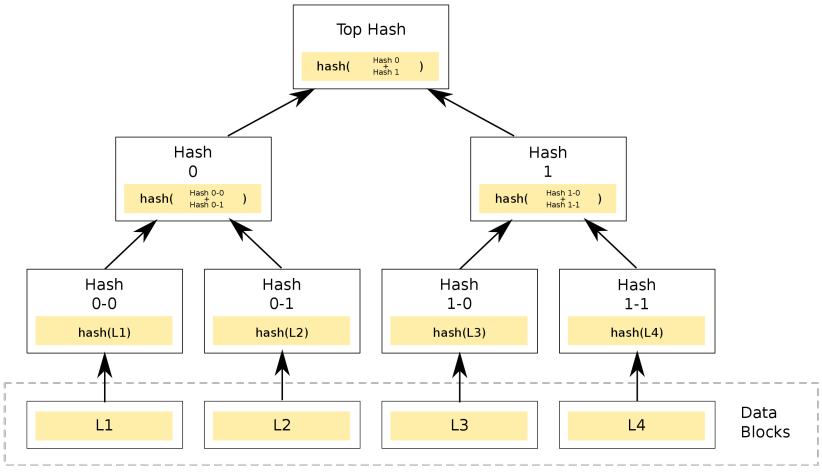

Figure 1.1: Binary merkle tree

The binary merkle tree consists of hashing the transaction two by two:

1. List all the data you want to save in an ordered list: it can be n users for example or any kind of data. In blockchain, these data are the transactions. Here say we have 4 transactions L1, L2, L3,

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 two obtain the parent. Here we obtain $Hash_0, Hash_1$.

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 you change 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)

To better understand we'll take an example of a bitcoin merkle tree for the block 170861:

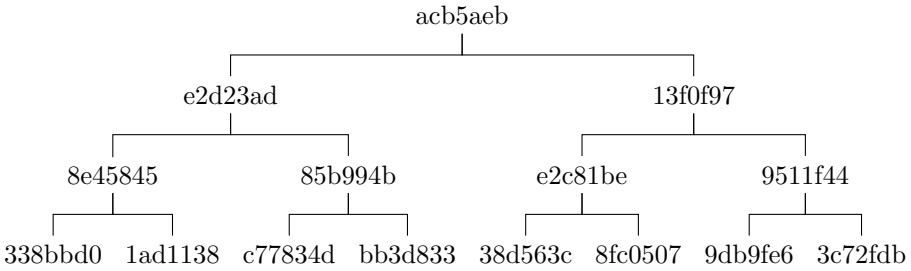

Figure 1.2: Merkle tree of the bitcoin block 170861

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, data is the transactions, and this structure allows other users to answer this question:

Is this transaction really included in that block?

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

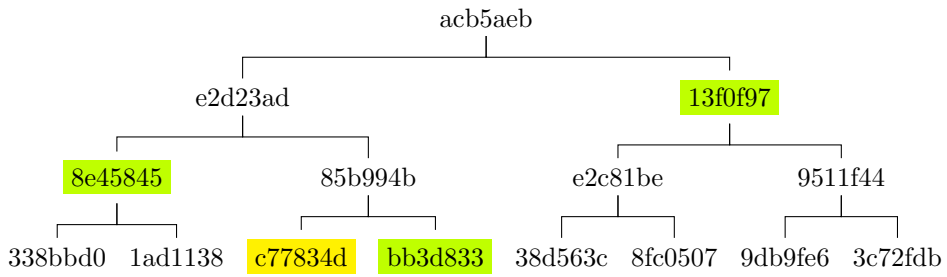

Figure 1.3: Proof of inclusion: Transaction to be verified in yellow, merkle proof in green

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 destinated to our account. We want a proof that this transaction has really happened, and have been included in that 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 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 they don'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 proof 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 childs and the binary tree from bitcoin, we can have 16 childs like in Ethereum. This has a drawback as it increase the length of the proof.

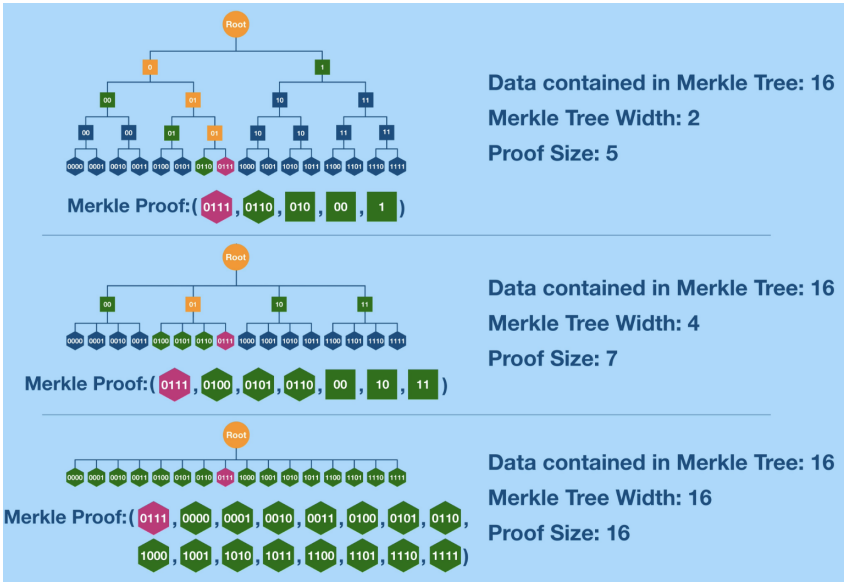

Figure 1.4: Merkle tree proof scaling

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

In general Verkle trees [9] 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 white paper use the merkle trees to represent data. This allow data integrity as for any distributed systems but also allow the light node to just download the fingerprint of the data.

[7] manage 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 the twitter accounts where the value is the number of followers, or the github contributors where the value is the number of commits.

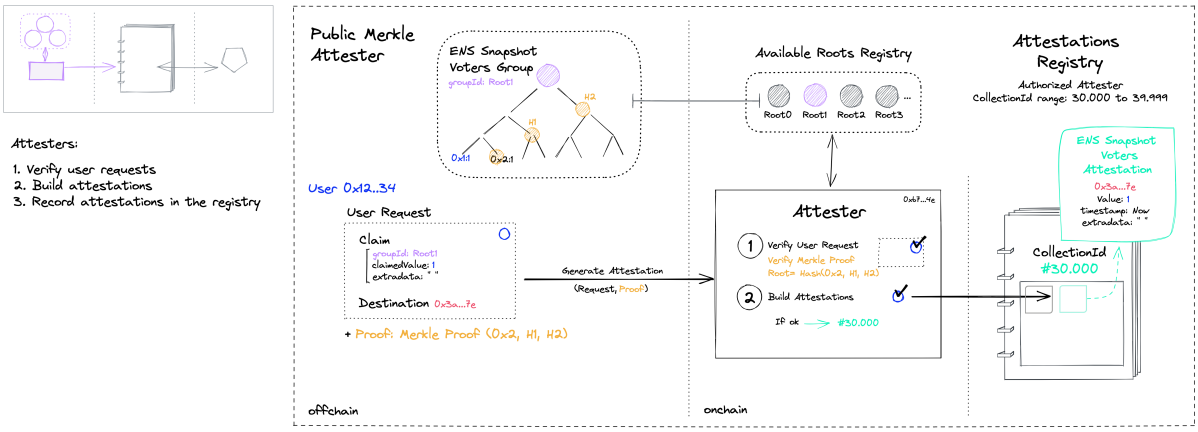

Figure 1.5: Sismo: user claims their attestation.

1.2 Blockchain basics

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

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

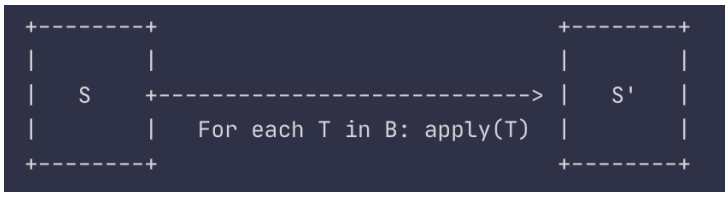

Figure 1.6: The state S transitioning to state S' by executing all transactions T from the block B

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.

1.2.1 The block

So what is stored exactly in a block?

Let's say for the moment that the 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.

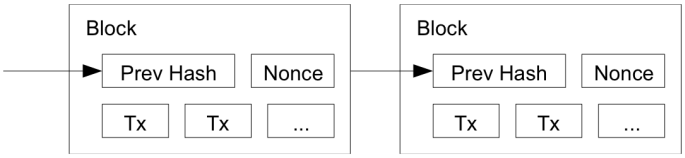

Figure 1.7: A bitcoin block: the hash of the previous block, the nonce, and the list of transactions

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

1.2.2 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, receipts.

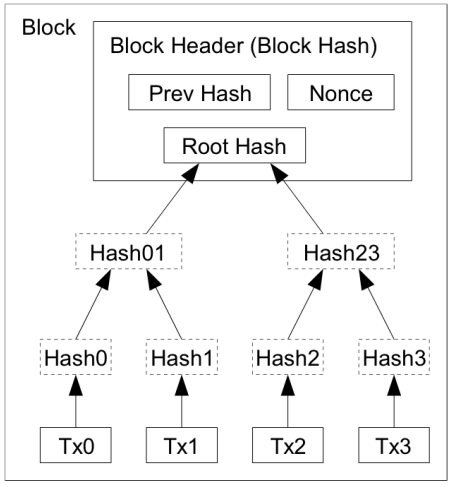

Figure 1.8: A Bitcoin[5] block

1.2.3 Single payment verification

This allows two actors to emerge:
The 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.
The light node asks for the blocks headers to the full nodes of the network. Once he got the block headers, he can ask the full nodes of the network to provide a proof for the transactions is interested in. With the procedure described in 1.1.2, the light node can verify that the transaction has indeed happened in the blockchain.

The 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.

1.2.4 Security of light client

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?

1.2.4.1 Proof of work

In bitcoin and the proof of work, light clients ask the full nodes for the last block headers and select the longest chain. A block in bitcoin and in 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 which make the hash of the block (including the nonce) starting with a certain number of zeros. In other words, they have to brute force a nonce such that the block hash meet the required level of difficulty: the number of zero. 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 probably 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.

1.2.4.2 Proof of stake

If the proof of work relies on the amount of work for 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, the 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, a verified website for example.

Interoperability

Merkle proofs from 1.1.2 are used to prove to a light client that a transaction has happened.
Now this can 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, [10] 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

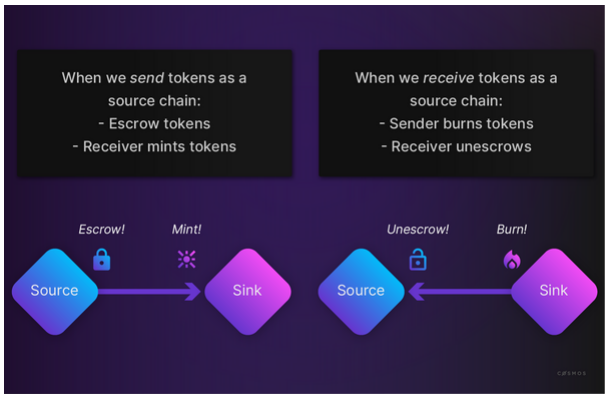

Figure 2.1: Token transfer (from cosmos)

1. The user from 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 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 a 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 A.1.3 there are three trees saved. One for the transactions as in bitcoin, one for the state, and one for the receipts.

These merkle trees allows Ethereum to answer more questions than just the inclusion of a transaction.

For example:

- Has this transaction been included in a particular block?

- What is the current balance of my account?
- Does this account exist?
- Tell me all instances of an event of type X (eg. a crowdfunding contract reaching its goal) emitted by this address in the past 30 days
- 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, 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 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 1.2.4, we highlight that the security of the light client 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 [12]

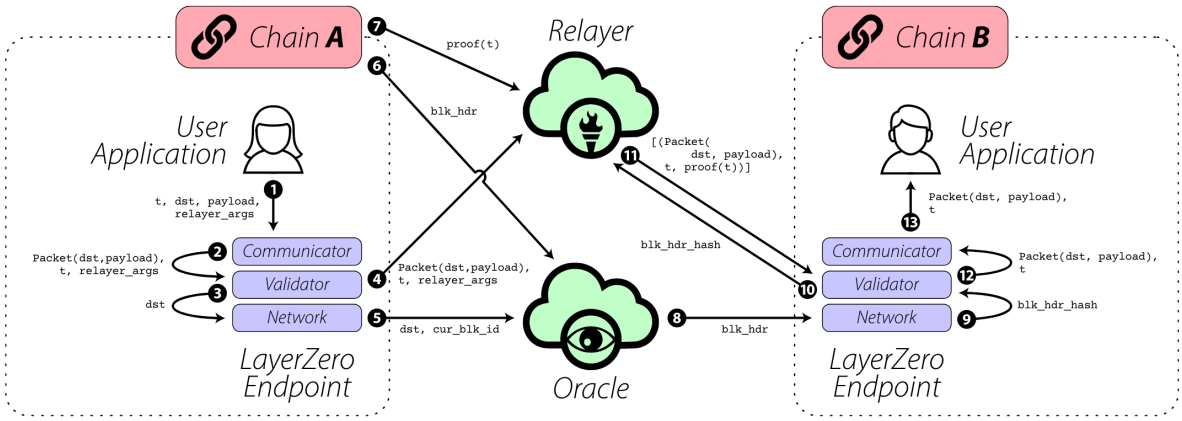

Figure 2.2: LayerZero

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 1.1.2, to verify that a transaction has taken place we need the merkle root, included in the block header and the 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 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 ZKBridge [11]

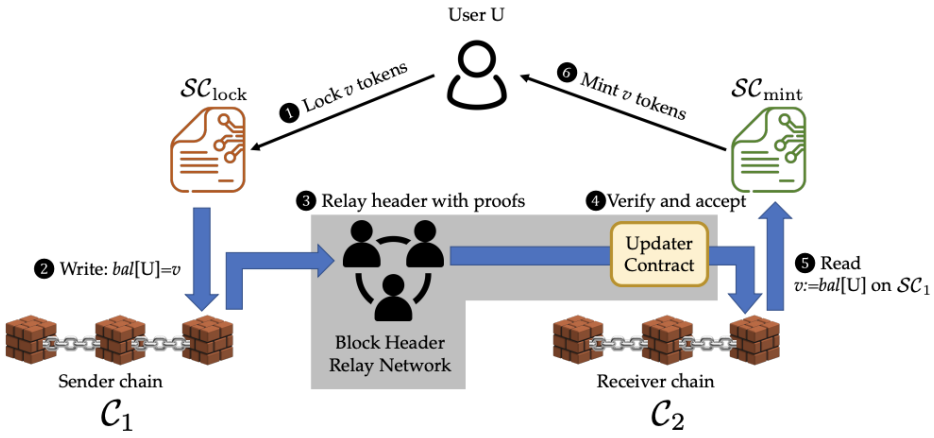

Figure 2.3: ZKbridge

Zero knowledge proofs are introduced here: A.2.

So here, the gray part is ZKbridge.

The most important part is done here by the relayers.

1. The relayers contact the full nodes of blockchain C1 to get the last block header
2. the relayers generate a zero knowledge proof that a light client of the sender chain C1 accepts that block header
3. This proof is then sent to the ZKBridge updater contract of the receiving chain C2
4. this updater contract verifies the ZK proof and adds it to its list of block headers if successfully verified.
5. The application contract, anyone on C2 can call the public function GetHeader from the updater contract to get the header.

Zk bridge achieve interoperability by verifying the validity of the block header. The relayers do that in a indirect way : they prove by a zk proof that a light client of the source chain accepts this block header.

Let's understand what the acceptance of the light client means. In ethereum, it means that:

- the status of the block header is finalized. Finality means that you can be sure it'll not be reverted.
- the sync committee, group of validators, have sufficient participants

- the aggregated signature is correct, meaning that the aggregator in Ethereum received enough votes and signed it.

In [3], you can directly check the code for validating the light client update, the acceptance of the block header.

Protocol 1 Block header relay network

procedure RELAYNEXTHEADER(LCS_{r-1},blkH_{r-1})
Contact k different full nodes to get the block headers following blkH_{r-1}, namely blkH_r.
Generate a ZKP π proving

$$\mathsf{LightCC}(\mathsf{LCS}_{r-1},\mathsf{blkH}_{r-1},\mathsf{blkH}_r) \rightarrow \mathsf{true}.$$

Send $(\pi,\mathsf{blkH}_r,\mathsf{blkH}_{r-1})$ to the updater contract.
end procedure

Figure 2.4: Relayer protocol

The security of the ZKbridge relies notably on the security of the light client of C1. It's because the zero knowledge proof consists in proving that this light client accepts the block.

2.2.3 Telepathy [8]

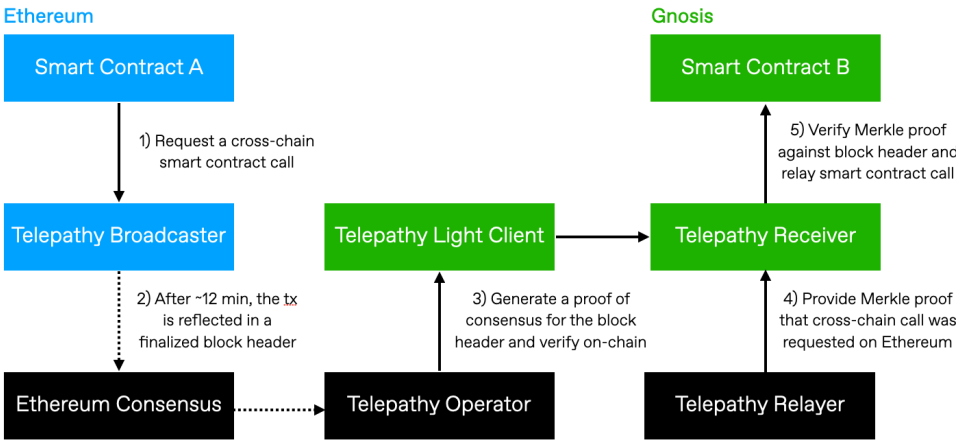

Figure 2.5: Telepathy workflow

Telepathy is based on proof of consensus. 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. More precisely, the 2.0 Ethereum has a sync committee of 512 validators randomly chosen every 27 hours and

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.

But verifying all these signatures one by one is computationally expensive. Instead the proof 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] *Light client update validation*. https://github.com/ethereum/consensus-specs/blob/dev/specs/altair/light-client/sync-protocol.md#validate_light_client_update.

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

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

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

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

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

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

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

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

Appendix

This part is placed in the appendix because it adds complexity and is not absolutely necessary to understand the main ideas on the interoperability.

A.1 Merkle trees

A.1.1 Trie

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.

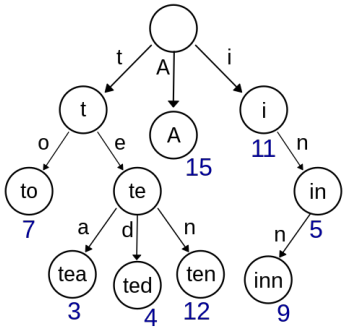

Figure A.1: Trie representing the words "A", "to", "tea", "ted", "ten", "i", "in", and "inn"

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 is 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, 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.

A.1.2 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 tree and Patricia tree are a compressed version of the above trie.

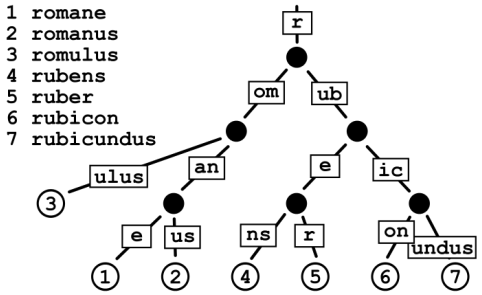

Figure A.2: Patricia tree

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

A.1.3 Patricia merkle tree

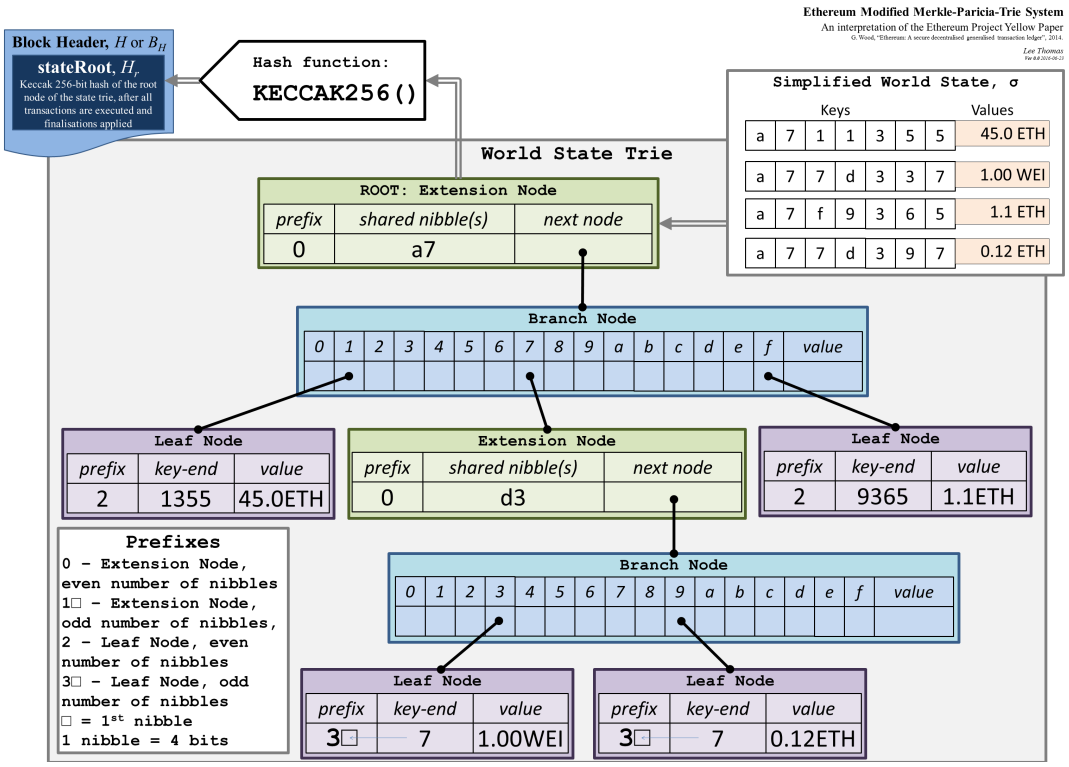

Figure A.3: Ethereum patricia tree

The Patricia merkle tree used in Ethereum modifies the original Patricia tree. 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 adress of the ethereum account and the value is the balance.

The extension node is the compression offered by the Patricia trie. It represents the shared nibbles (letter) 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.

A.2 Zero knowledge

This is meant to be a light introduction to the zero knowledge proof concept to understand the following 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 digital signature, you send a signature proving that you are indeed the one sending the mail. More precisely, this signature prove that you know the private key without revealing it. The verifier checks that the signature matches the public key of the sender. ($\sim$ 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 when you send H=SHA3(m) 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 ($a_{n-2} + a_{n-1} = a_n$)
- the transactions are valid. This allow the ZK roll-ups to process the transactions off chain, and prove on chain to layer 1 that these transactions are indeed valid.

In fact, zero knowledge proofs allow two properties.

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

Privacy with zero knowledge, you don't reveal anything about the statement. In Zerocash[6], the privacy is needed to ensure the confidentiality of the transactions.

A zero knowledge protocol should verify:

- **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 of short proof in terms of size for example. We also want, and this is a fundamental requirement, a very fast verification. 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 one.

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