The First Amendment White Paper

Authors:
Nebojsa Konstantinovic and Bojan Jovin,
Merkle Blue DOO

Table of contents

1 Abstract

Centralization of information publishing platforms comes with a wide variety of content control mechanisms, ranging from direct censorship to more subtle ones such as controversial company policies that favor certain types of publishers. For instance, Youtube has recently issued the "demonetization" policy around content that could be perceived to be "hateful or inflammatory". IMDb has suspended their message board system since it is "no longer providing a positive, useful experience for the vast majority of users". Many of the service rating websites have been known to remove negative client reviews, and the list goes on.

As the internet is shifting from centralized proprietary services towards decentralized, open ones, **The First Amendment** project aims to utilize this trend to tackle some of the major issues present today, such as content censorship, author credibility, and accountability.

The First Amendment aims to deliver a decentralized, trustless, censorship-resistant content publishing platform that runs on Ethereum blockchain combined with IPFS content storage system.

This document:

1. Gives an overview of the current state of The First Amendment project, referencing the technologies and system architecture used.
2. Presents a **Layer 2** solution proposal that tackles scalability issues and fees unpredictability problem.
3. Gives a presentation of the team behind the project.

2 Current System Overview

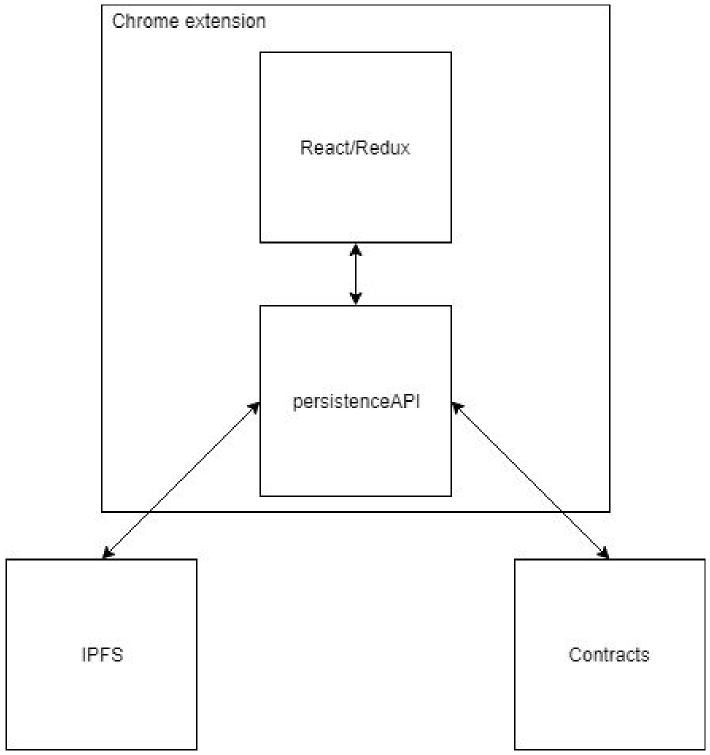
In its current form, The First Amendment (TFA) functions as a Chrome extension that allows users to comment on any topic on the web freely. As described in the Abstract, the project aims to evolve into a decentralized, trustless, censorship-resistant content publishing platform and, to accomplish these features, it utilizes the Ethereum blockchain in combination with IPFS. IPFS (described in greater detail in chapter **2.1**) is used to store user content, and provide content hashes as its unique identifiers, whereas the Ethereum blockchain is used to anchor those IPFS hashes to the blockchain, thus providing the mechanism guaranteeing that the history of the posts is intact. The combination of two (blockchain and IPFS) is already used in many projects (for example, Filecoin). Since the platform is blockchain based, it inherits its desired properties:

- **Censorship resistance** - no one can change/delete content, or ban users.
- **Trustlessness** - no trusted third party. Users can verify network history by themselves.
- **Resilience** - there is no single point of failure. The system as a whole is resistant to attacks.

In addition to that, IPFS provides:

- **Network neutrality** - enables censorship resistance.
- **Content immutability** - guarantees content integrity.

Properties listed above make up enough of building blocks for the TFA platform. Persistence API is a TFA javascript library that uses these building blocks to perform actions such as persisting a new comment, upvoting, downvoting, creating a persona (user account), et cetera. It also builds a data model and feeds it to the presentational layer of the application.

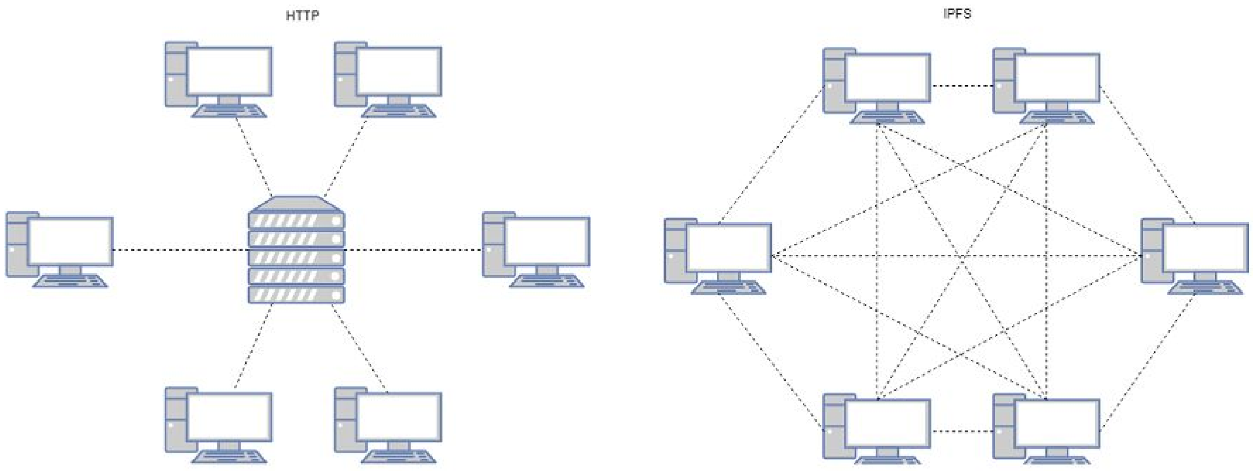


TFA current system building blocks

2.1 IPFS

2.1.1 What is IPFS

Interplanetary File System (IPFS) describes itself as "a peer-to-peer hypermedia protocol to make the web faster, safer and more open." In its essence, it is a peer-to-peer distributed file system that connects computing devices with the same system of files. It can be seen as a high throughput content-addressed block storage model, which provides content addressed hyperlinks, with no single point of failure, and where nodes do not need to trust each other. [Benet]



HTTP vs. IPFS

IPFS solves the problem of content addressing, by using content-addressable storage as opposed to location-based content storing, widely used on the web today. IPFS network addresses the content by its hash, thus providing **content integrity**. Subsequently, any node can store any content, and a consumer can be sure that content he received from the IPFS network is indeed the content he requested, just by verifying that received content hash matches the requested address.

Network neutrality and the decentralized nature of IPFS make it censorship resistant, and content addressing system guarantees data integrity. This means that no one can interfere with content published by The First Amendment users.

2.1.2 IPFS and The First Amendment

As previously mentioned, IPFS is used as a storage system for TFA comments and user profile attributes. Any data stored on IPFS gets assigned a hash address which is subsequently anchored to the blockchain ledger (more on this in chapter 2.2). This hash address is permanent and immutable, meaning that it ensures that nobody could alter the content of the published data. Any attempt to do so would alter the content hash, rendering it unreachable, since the new hash would not be found on the blockchain.

2.2 Ethereum

Ethereum is a blockchain-based platform that executes decentralized smart contracts in Ethereum Virtual Machine (EVM) on multiple network nodes and uses proof of work to establish a consensus on the state of the network. Smart contracts are Turing-complete pieces of code that alter the state of the blockchain. One could view Ethereum as a distributed global computer where smart contracts are executed. Every operation that gets executed inside the EVM is simultaneously executed by every node in the network. Once the consensus on the blockchain state is reached (when transactions are mined), the state of the blockchain is irreversible (hard-forks are out of the scope of this document).

This state immutability is a vital blockchain feature from TFA perspective as it guarantees that all content published on the blockchain cannot be deleted or altered. On the other hand, all the data published on the blockchain is transparent, which means that it is visible to anyone at any time.

2.2.1 Ethereum and The First Amendment

Smart contracts are immutable programs that execute when certain conditions are met. These smart contracts can be scripted with a set of instructions on the virtual machine which are executed in an immutable and **transparent** fashion.

TFA currently has two contracts:

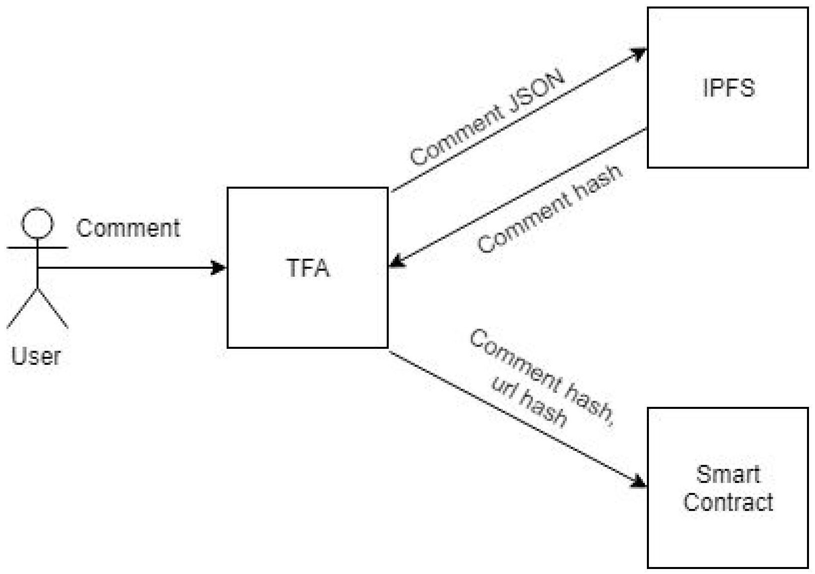
- **Comment** - responsible for adding, upvoting and downvoting user-submitted comments.
- **Persona** - responsible for associating certain user attributes (such as username and profile image) to a specific Ethereum account.

2.2.1.1 Adding a comment

addComment(urlHashRef, commentHashRef)

Once a user submits his comment using TFA application, it gets stored to IPFS as a JSON file. In return, IPFS produces a specific unique hash that matches the uploaded file. TFA then makes a call to Comment contract on Ethereum, sending it the received hash coupled with the hash of the URL that the comment is associated to. The contract's only function is to emit an event containing the two hashes. Once the transaction is mined, transaction hash specific to that function call (and therefore to that comment) is generated, and the event is preserved in the transaction log. TFA application can further filter out this comment based on its:

- transaction hash,
- URL hash,
- uploader address.



System components interactions

2.2.1.2 Upvoting a comment

upvoteComment(txHashRef)

Once the user upvotes the comment, TFA submits a call to Comment contract with comment transaction hash as a parameter. The contract then emits an Upvote event that gets preserved in the transaction log, once the transaction is mined. TFA can then acquire the number of comment upvotes by parsing the Upvote events associated with a specific comment transaction hash.

2.2.1.3 Downvoting a comment

downvoteComment(txHashRef)

Similarly to upvoting, once the user downvotes the comment, TFA submits a call to Comment contract with comment transaction hash as a parameter. The contract then emits a Downvote event that gets preserved in the transaction log, once the transaction is mined. TFA can then acquire the number of comment downvotes by parsing the Downvote events associated to a specific comment transaction hash.

2.2.1.4 Adding a persona

setAttributes(IPFSHash)

Once the user submits a new persona, TFA stores the persona attributes to IPFS as a JSON file. Just like with comments, IPFS then produces a hash that matches the stored file. This hash is then associated to the uploader account address, by the Persona contract.

2.2.1.5 Updating a persona

setAttributes(IPFSHash)

Updating a persona uses the same logic as adding a persona, where a new set of attributes get associated to an already existing uploader account address.

2.3 Persistence API

Persistence API is The First Amendment javascript library that ties together IPFS and Ethereum. It exposes functionality such as:

- addComment,
- getComments,
- upvoteComment,
- downvoteComment,
- addPersona,
- getPersonas,
- updatePersona,
- getAccounts,
- getAccountBalance, et cetera.

Persistence API performs the above-listed actions by interacting with IPFS javascript API and Ethereum web3 javascript API. Apart from that, it has the responsibility to build the data model to be used by representational components (described in chapter 2.4).

2.3.1 Assembling comments

When a user visits a certain page, TFA creates a hash of its URL, and makes a query on the blockchain, requesting all of the AddComment events persisted in the transaction log that matches the generated hash. After receiving all of the comments coupled with the requested URL, it acquires three pieces of information of interest for each comment: comment transaction hash, comment uploader account address and comment content IPFS hash.

Using the comment transaction hash, it produces another query for each received comment, where it gathers and counts all of the Upvote/Downvote events associated to that Tx hash.

Using the uploader account address of each comment, it retrieves user attributes associated to that address.

Finally, using IPFS hash, it retrieves the content of the comment, stored on the IPFS.

Once all of the above is gathered for each comment, the TFA persistence API has all the information it needs for building Comment objects, later to be distributed to and used by the presentational layer.

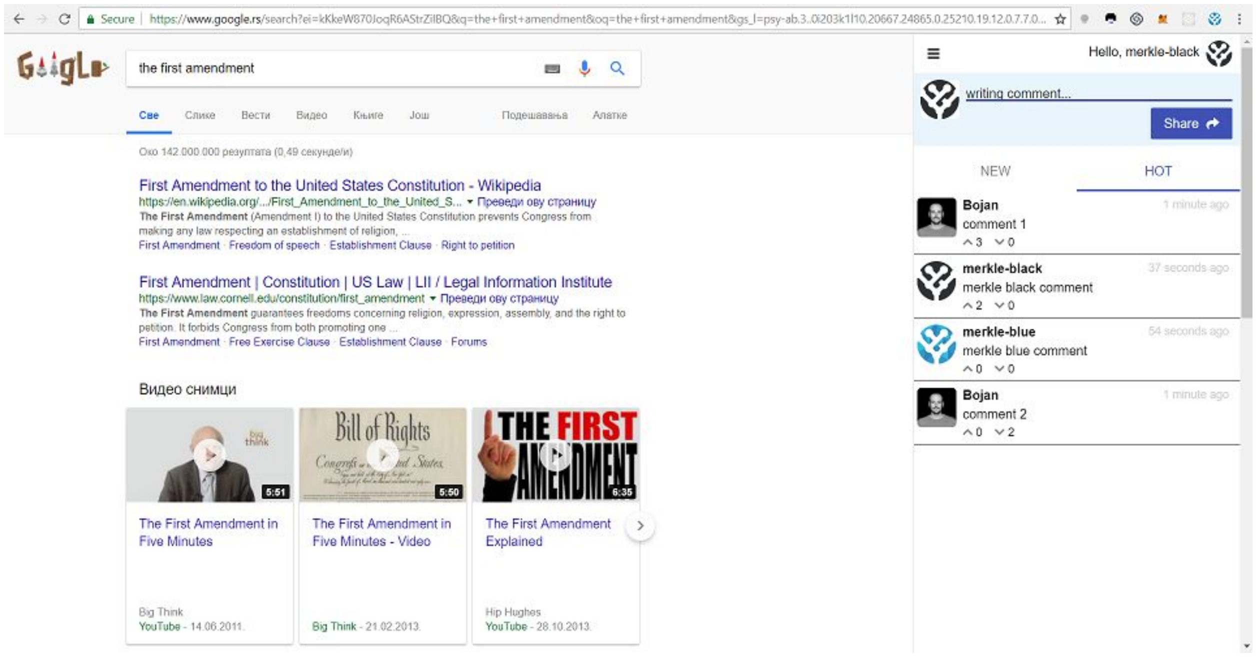
2.4 UI

The user interface is currently a Chrome extension built using React and Redux, React being a presentational layer, and Redux serving as a model container. As TFA inclines to become a platform (which implies building a Mozilla add-on, a website, Android and iOS app, and an API for native websites integration), React/Redux combination provides flexibility regarding components reusability.

As a Chrome extension, TFA presents its data in the form of a sidebar that gets injected into the native page. The sidebar currently has two panels:

- Comments management panel
- Persona management panel

The comments management panel displays all of the comments associated with the currently visited page and allows users to leave new comments.



Око 142.000.000 резултата (0,49 секунде/и)

First Amendment to the United States Constitution - Wikipedia
https://en.wikipedia.org/.../First_Amendment_to_the_United_S... ▾ Преведи ову страницу
The First Amendment (Amendment I) to the United States Constitution prevents Congress from making any law respecting an establishment of religion, ...
First Amendment · Freedom of speech · Establishment Clause · Right to petition

First Amendment | Constitution | US Law | LII / Legal Information Institute
https://www.law.cornell.edu/constitution/first_amendment ▾ Преведи ову страницу
The First Amendment guarantees freedoms concerning religion, expression, assembly, and the right to petition. It forbids Congress from both promoting one ...
First Amendment · Free Exercise Clause · Establishment Clause · Forums

Видео снимци

The First Amendment in Five Minutes — Big Think, YouTube - 14.06.2011 (5:51)
The First Amendment in Five Minutes - Video — Big Think - 21.02.2013 (5:50)
The First Amendment Explained — Hip Hughes, YouTube - 28.10.2013 (6:35)

Bojan — 1 minute ago
comment 1
^3 v0

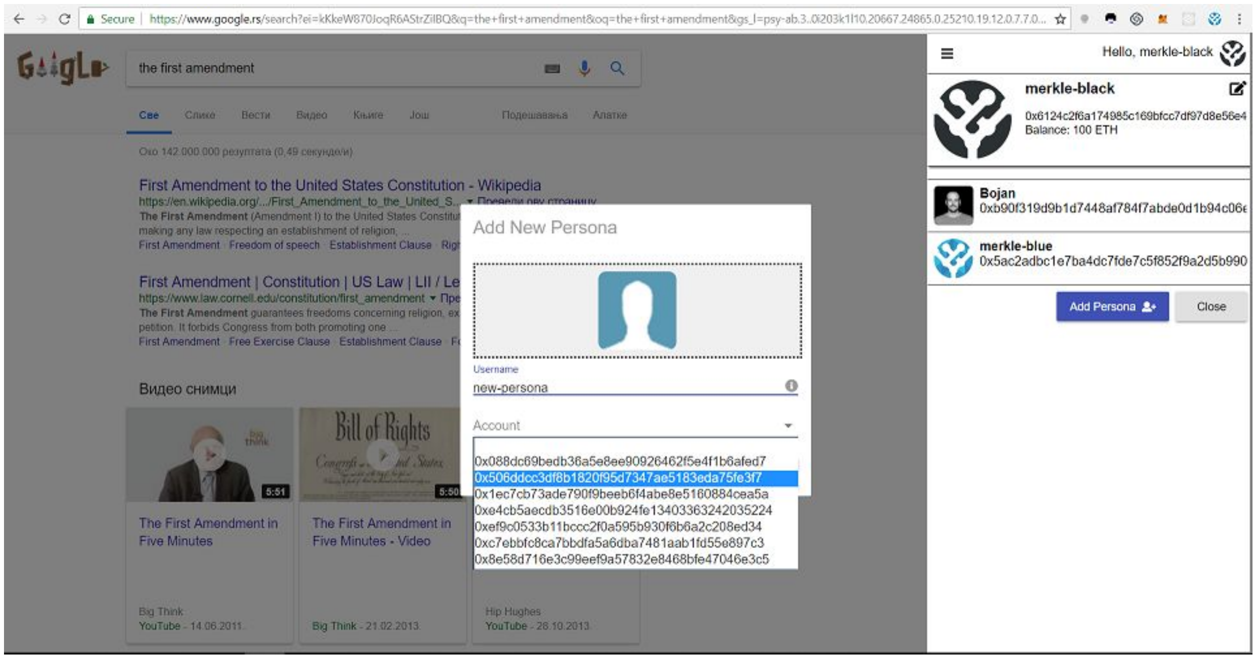
merkle-black — 37 seconds ago
merkle black comment
^2 v0

merkle-blue — 54 seconds ago
merkle blue comment
^0 v0

Bojan — 1 minute ago
comment 2
^0 v2

Comments management panel

Persona management panel allows adding and updating personas and selecting a current persona.



Hello, merkle-black

merkle-black
0x6124c2f6a174985c169bfcc7df97d8e56e4
Balance: 100 ETH

Bojan
0xb90f319d9b1d7448af784f7abde0d1b94c06

merkle-blue
0x5ac2adbc1e7ba4dc7fde7c5f852f9a2d5b990

Add New Persona

Username
new-persona

Account

0x088dc69bedb36a5e8ee90926462f5e4f1b6afed7
0x506ddcc3df8b1820f95d7347ae5183eda75fe3f7
0x1ec7cb73ade790f9beeb6f4abe8e5160884cea5a
0xe4cb5aecdb3516e00b924fe13403363242035224
0xef9c0533b11bccc2f0a595b930f6b6a2c208ed34
0xc7ebbfc8ca7bbdfa5a6dba7481aab1fd55e897c3
0x8e58d716e3c99eef9a57832e8468bfe47046e3c5

Persona management panel

The current system is tested and works on private testnet (Ganache) and Ropsten testnet. However, it has some limitations described in the following chapter.

3 Current System Limitations

Every user's post/action needs to be a valid Ethereum transaction (Tx) to be included in the blockchain. After the Tx is mined, other users can be sure about who posted content (address or identity) and that the post is paid for (not spam). These properties are highly desirable but can be costly to a user. The following are considered to be limitations of the current system:

- **Miner fees** - unpredictable and expensive.
- **On-chain scaling** - a limited block capacity might impose a scaling issue for the platform.
- **Latency** - content is available only after it is mined.

3.1 Layer 2 Solutions

The system can be moved to a second layer to overcome current limitations. Second layer solutions significantly reduce miner fees and make them more predictable, while vastly improving scalability. Therefore, all the desired properties of a blockchain are still preserved with one additional property enabled: users do not need to wait for posts (Txs) to be mined. They can consume content nearly instantly. There are various approaches to implementing second layer solutions in the blockchain ecosystem, most of which utilize off-chain transactions that get anchored to the main blockchain to ground its security. Most notable of these solutions are state channels (Lightning network, Raiden, uRaiden) and child chains (Plasma).

3.2 Payment channels

Most prominent payment channels solutions on Ethereum platform are uRaiden and Raiden (developed by Brainbot technologies). Their purpose is to reduce fees and increase throughput between two parties that interact frequently. For example, uRaiden is based on a uni-directional payment channel between a sender and a receiver. For the sake of demonstration, let us assume that Alice and Bob want to transact using uRaiden. Alice is Bob's customer and makes frequent purchases at Bob's store. Alice opens a payment channel to Bob by publishing a transaction Tx to Ethereum network that deposits Alice's funds to smart contract. Once the Tx is mined, the payment channel is open, and Alice's funds are locked in a smart contract for future spending. Alice can now send Txs to Bob off-chain, through their private communication channel, usually using HTTP request. Each transaction in a channel contains a payment channel balance update and Alice's signature. Balance update contains information on how much of Alice's deposit is sent to Bob. Bob receives Txs and stores them. When Alice and Bob decide to close the payment channel, Bob signs the last Tx and publishes it to the Ethereum network. Once the Tx is mined, the channel is closed, and Alice's deposit is sent to both Alice and Bob, where Bob receives the spent part of it, and Alice receives what remains. Alice can top-up the deposit at any time while the payment channel is open.

3.3 Payment channels Limitations for TFA use case

Payment channels are limited to two parties only. No third party can observe what is happening in the channel except for Alice and Bob since the channel is private. If we would like to add observers to a channel, those observers could close the channel at any time since messages sent through the channel obtain information needed for channel closing. Channel closing is costly for both Alice and Bob, and no one can detect which of the observers closed the channel.

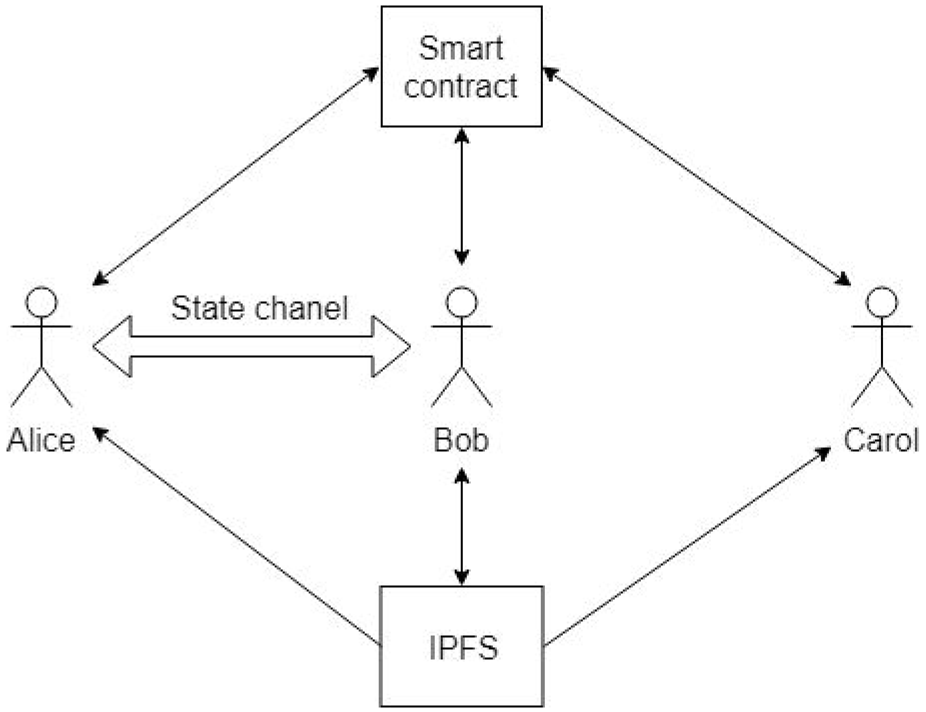
Also, in the case of bi-directional payment channels, there is a constraint that Alice has to be online during the settlement period. Otherwise, Bob can close the channel with a transaction that is more favorable for him, without being corrected. TFA layer 2 solution does not have this kind of a constraint as Bob's competitors, and system auditors always make sure that the channel is closed correctly.

4 TFA Layer 2 Solution Proposal

TFA second layer solution is based on state channels, very similar to uRaiden channels (explained in more detail in 3.2). Users in a system open channels to content delivery providers to be able to post content on the network. Channels are used to send signed messages (Txs) containing the user activity on the network. Interested parties can get channel updates nearly instantly. Rules of the system are coded in smart contracts, and users are incentivized to follow the rules. Malicious users who break the rules get penalized.

4.1 Actors

There are three primary actors in the system: Alice, Bob, and Carol. Each of them has a distinctive role. There is no restriction on the number of each of these actors in the system. All actors interact with smart contract and IPFS.



Actors in the system

4.1.1 Bob

Bob serves as a content delivery node and payment processor. He makes a profit from comment fees but also holds the most significant responsibility in the system. Users open channel to Bob to be able to post content, whereas Bob has a responsibility to maintain the valid data model and to communicate to other Bobs in the system to ensure overall system validity. Before he can deliver content and process payments, Bob must submit one large deposit to a smart contract which he could lose if any third-party catches him breaking the rules. For each opened channel, he must top the deposit up with a smaller amount (to ensure that Bobs with more customers hold more substantial responsibility). As long as he follows smart contract defined rules, he does not lose the deposit. Bob can have multiple users with open channels to him but has only one large deposit.

Bob is interacting with:

- **Alice** - he has an open channel with Alice and processes her transactions (comments, upvotes, downvotes).
- **Smart contract** - watches the state of smart contract, submitting transactions and actively participating in disputes.
- **IPFS** - he publishes content on the IPFS, and also reads content from it.

4.1.2 Alice

Alice is an author/user who is an active participant on the platform. She has an open channel with Bob with a committed deposit to be spent (similar to uRaiden channel). These funds are used to pay for posts, upvotes and downvotes she makes on the platform.

Alice is interacting with:

- **Bob** - she has an open channel with Bob.
- **Smart contract** - watches the state of smart contract and can submit channel closing transactions and disputes.
- **IPFS** - she is subscribed to Bob's IPFS repositories and also reads objects from IPFS.

She can additionally store content on the IPFS for better content delivery.

4.1.3 Carol

Carol is an auditor and only reads content. She does not have an open channel with anyone, but she is also checking if Bob is breaking any rules. If she can prove that Bob is cheating, she can withdraw a part of Bob's deposit from the smart contract. Carols are usually parties whose incentive is for the system to work correctly, such as other Bobs, or stakeholders, but basically, anyone can perform the tasks of this role. Carol needs to submit a small one time fee to a smart contract, to be registered as the system auditor.

Carol is interacting with:

- **Smart contract** - watches the state of the smart contract, and can submit a dispute.
- **IPFS** - she is subscribed to Bob's IPFS repositories, making sure that Bob's activity is following the smart contract defined rules.

4.2 Smart Contract

The smart contract is deployed on the Ethereum network and is visible to all participants. State of the contract is visible to anyone. It has four primary functions:

- **Hold Bob's deposit** - since Bob is the most responsible party in the system (a trade-off for making profit) he is required to submit a large deposit on the smart contract, which can be spent by anyone who submits proof that he is breaking the rules.
- **Store open channels** - all of the open channels used to transact on the second layer.

- **Defines rules** - minimum fee value, rules dispute, et cetera.
- **Store registered Carols** - store information on all of the system auditors that can participate in disputes.

4.3 Storage system

IPFS is used to store a data model in the system. As an object is stored on IPFS, its hash represents a unique identifier used to access that specific object. IPFS hashes are anchored on blockchain and used to fetch data from the IPFS after the state channels are closed.

4.4 Channels

Alice opens a channel to Bob with committed deposit, where an opening transaction is being published to the blockchain. After the opening transaction is mined, the state channel is considered open, and Alice and Bob can communicate by sending each other messages. Messages are sent from Alice to Bob but never submitted to blockchain during the channel lifespan.

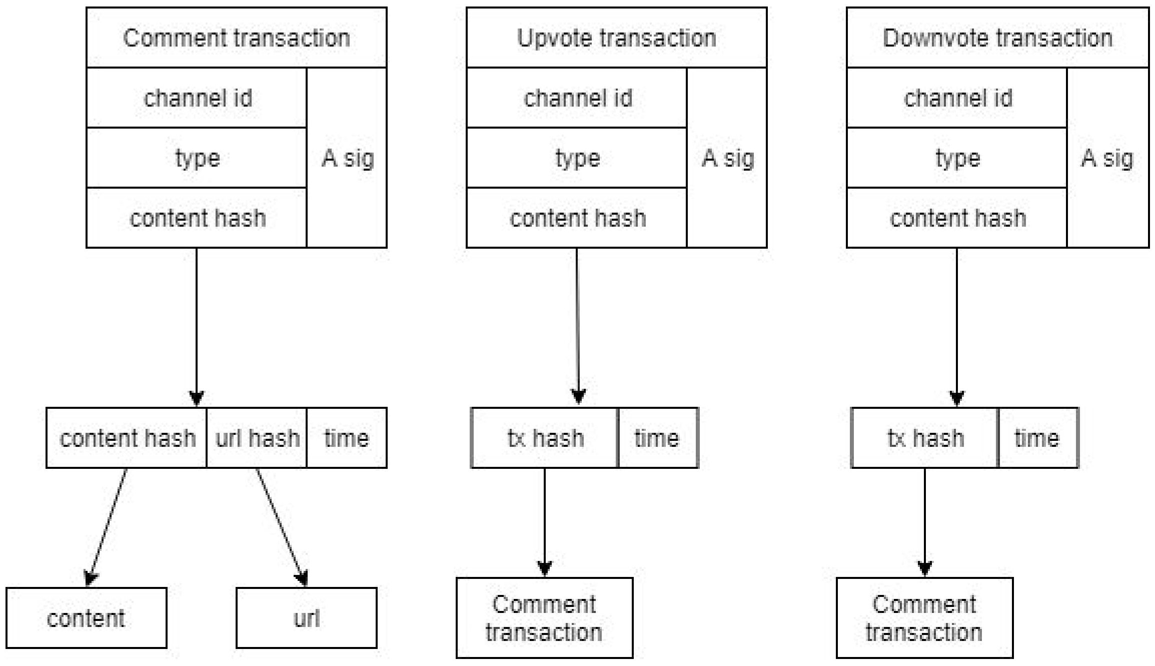
When Alice or Bob decide that they want to close a channel, they can publish the latest state of the channel and wait for it to get mined. The state of the channel is represented by an IPFS header object that points to a Merkle tree data structure that holds all of the transactions belonging to that specific channel. State update that gets published to Ethereum network closes the channel and represents its final state. After the final state update is mined, and there are no disputes, it becomes visible to all participants in the system. After the channel is closed, it cannot be reused in further system functioning. If the business relationship between the same Alice and Bob should revive again, a new channel has to be opened. Messages exchanged between Alice and Bob within a channel correspond to transactions that represent comments submission, comments upvoting or downvoting.

4.5 Transactions

Transactions in the system are regular Ethereum transactions that get published in the appropriate data structures, explained in more detail in the text to follow. Txs cannot close the channel but can be used for dispute, when presented with appropriate fraud proofs. To close the channel, Bob or Alice must provide the valid IPFS header object that points to a Merkle tree structure in which the transactions are published.

There are three kinds of transactions in the system:

- **Comment Txs** - representing the comment published by Alice.
- **Upvote Txs** - representing the upvote of some other comment Tx.
- **Downvote Txs** - representing the downvote of some other comment Tx.



Transactions

4.5.1 Tx structure

Every Tx contains the following data:

- **signature** - proving that Alice signed it,
- **channel id** - channel identifier,
- **type** - comment, upvote or downvote,
- **content hash** - hash of an IPFS object containing content, URL, and timestamp in case of a comment transaction, or the other Tx hash and timestamp in case of an upvote/downvote transaction.

A timestamp is a current block hash, and it is used to enforce that Bob and Alice cannot back-date or submit transactions with future dates.

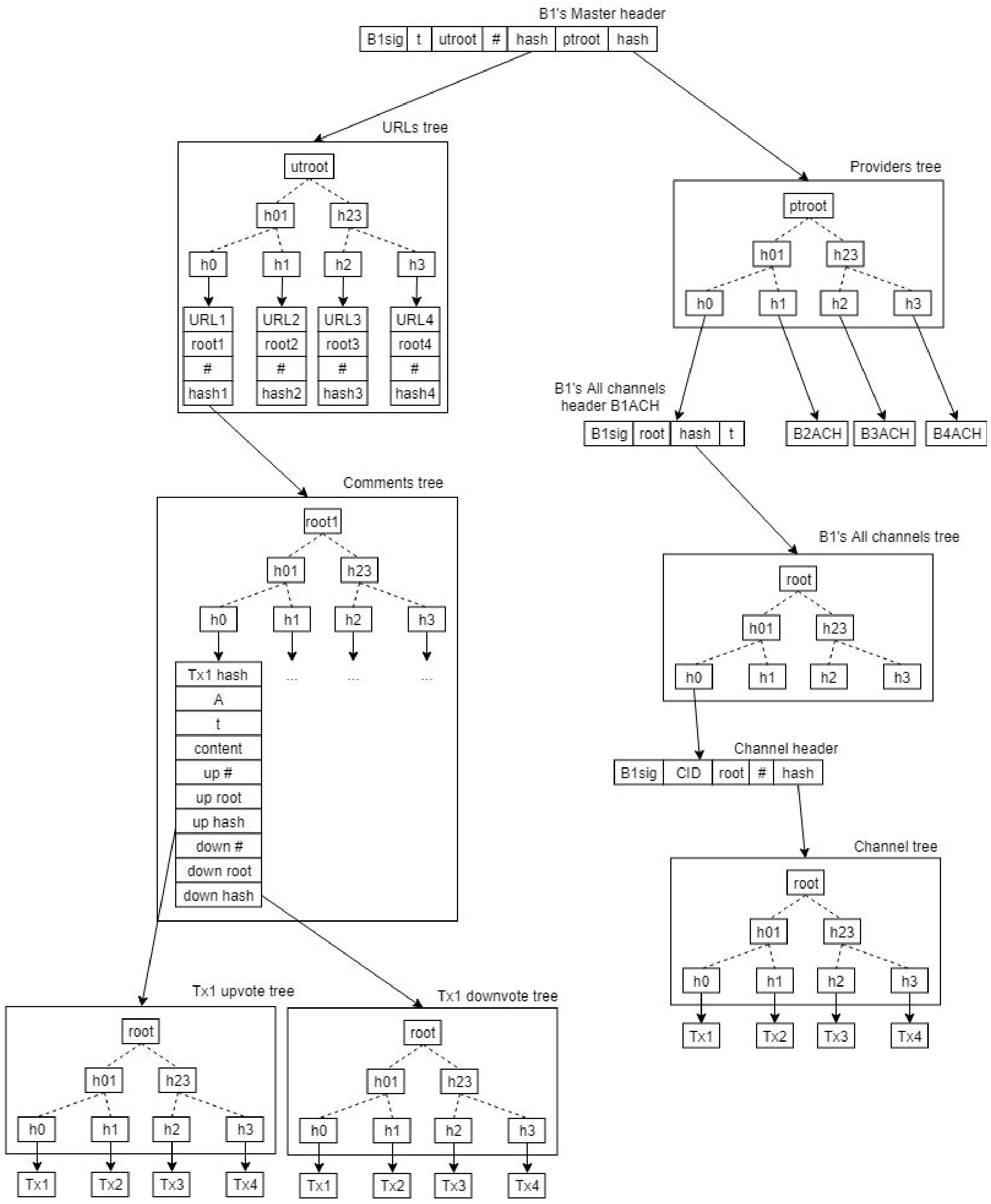
For simplicity sake, in this proposal POC, all transactions (comments, upvotes, and downvotes) have the same cost.

4.6 Logical Data Model

Logical data model consists of two distinct logical parts, presentational (left side of the picture below) and channel model (right side of the picture below). Master header ties both parts into one logical model. Both models traverse to transactions to the bottom, but in a different way. Presentational model is optimized to group data by URLs, so it is easy for readers to reach comments for a particular URL. Channel model groups data by channels, so that channel closing data structures are visible and easily verified.

All objects in the model are interconnected and inherently signed by Bob in the master header. All data is visible for everyone to verify, as every Bob must serve content. Data in two models must match at all times, meaning that all of the transactions included in the channel model must be included in the presentational one, and vice versa. If any discrepancy in the two models is detected, Carol can submit a dispute by presenting valid proof of that discrepancy.

All of the Merkle trees in the model must hash correctly, and all of the timestamps in the model are blockchain block hashes.



The master header is at the top of the logical data model, and contains the following data:

- **B1sig** - Bob1's signature. The signature must be valid.
- **t** - the time of the last update (Ethereum block hash).
- **utroot** - root element of the Merkle tree that contains the URLs tree. This root element must match the root of the URLs tree.
- **#** - the number of elements in the URLs tree. The number of elements must match the number of leaves in URLs tree.
- **uthash** - hash of the URLs tree IPFS object.
- **ptroot** - root element of the Merkle tree that contains Providers tree. This root element must match the root of the Providers tree.
- **pthash** - hash of the Providers tree IPFS object.

For simplicity sake, it is forbidden for two Master headers with the same timestamp to coexist. In later versions, we might introduce pointers to a previous Master header, for faster content delivery.

4.6.1 Providers tree

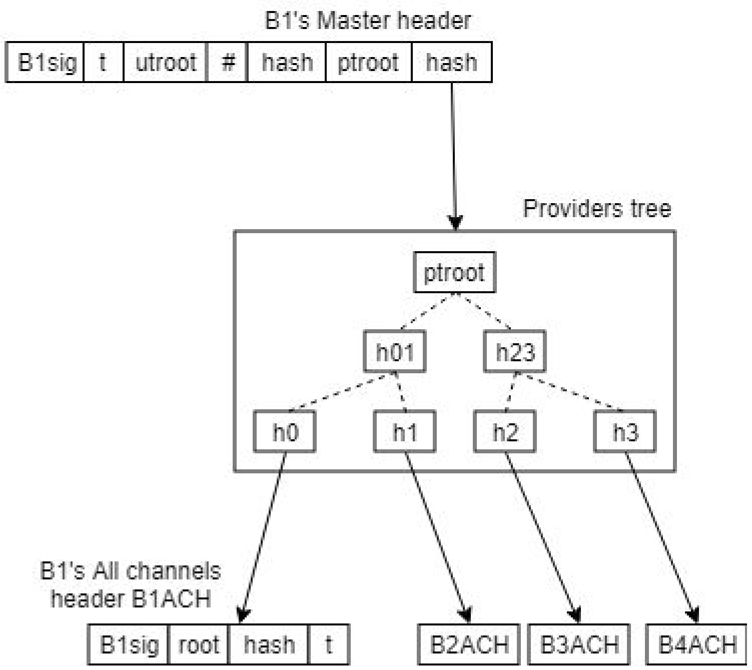
Providers tree is used for storing **All channels headers** from all Bobs registered in the system. Every Bob has his own "slot" in the Providers tree, based on his unique identifier. If Bob omits one of the All channels headers, he can be disputed. All channels headers in a tree must be verified objects, in a sense that no disputes can be called on them. In a case of a dispute, both disputed Bob and the one that is storing header linked to the dispute could lose their deposit. Therefore it is each Bob's responsibility to perform a validity check on the All channels header before he includes it in his Providers tree. Updates must be done regularly, meaning that the difference between a timestamp in the Master header and a timestamp in all of the included All channels headers must be less than a predefined time window. There are three exceptions to this:

- There is a fraud proof called on a specific All channels header.
- There is a fraud claim posted on a specific All channels header.
- There is a data unavailability claim posted on a specific All channels header.

The difference between a fraud proof and a fraud claim is that the fraud proof relates to a fraud that is provable to the Smart contract. Fraud claim refers to a fraud that cannot be proven to the Smart contract (usually due to the size of the proof) but can be made evident to all of the participants in the system.

We'll go into more details regarding fraud proofs, fraud claims and the data unavailability problem in chapters that describe the system functioning.

Also, adding an All channels header into the Providers tree must be done in chronological order (a new version of the tree cannot contain the header with a lower timestamp).



Providers tree

Providers tree's primary function is to be used as a truth discovery mechanism in a case of a dispute. The mechanism of dispute and truth discovery is explained in more detail in the Fraud proofs chapter.

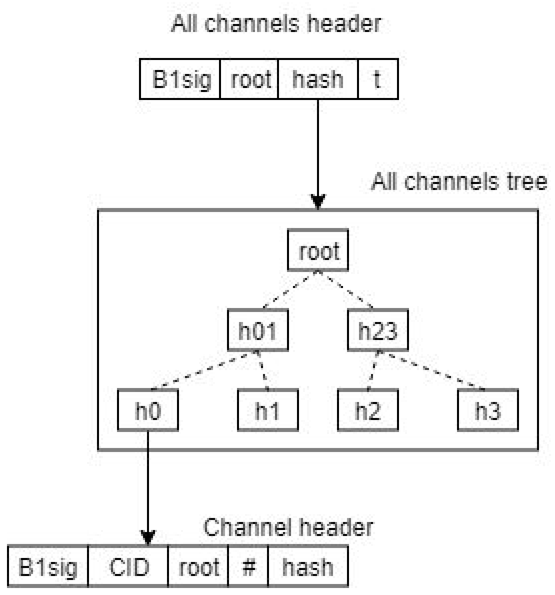
The number of elements in Providers tree must match the total number of Bobs in the system.

4.6.2 All channels tree

All channels tree is used for storing channel headers for all Bob's customers (Alices), in the form of a Merkle tree where Merkle tree leaves are pointing to channel headers.

Every channel has its own "slot" in the All channels tree, based on its unique identifier. If Bob omits one of the channels, he can be disputed.

For simplicity sake, it is forbidden for two All channels headers with the same timestamp to coexist. In later versions, we might introduce pointers to previous All channels header, for faster content delivery.



All channels tree

All channels header contain the following data:

- **B1sig** - Bob1's signature. The signature must be valid.
- **root** - root element of the All channels Merkle tree. The root element must match the root from the IPFS Merkle tree object.
- **hash** - hash of the All channels Merkle tree IPFS object.
- **t** - the time of the last update.

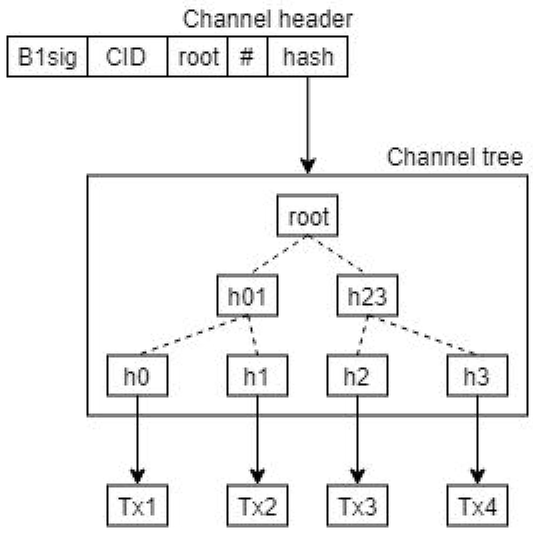
The number of elements in the All channels tree must match the total number of channels in the system.

4.6.3 Channel tree

Channel tree is used to group all of the transactions that belong to the same channel, in the form of a Merkle tree. Merkle tree leaves are pointing to transactions in the channel.

Timestamps of each Tx contained in the Channel tree must not exceed the timestamp of All channels header that contains that Channel tree. Also, no Tx added in a new iteration of Channel tree, should have the timestamp that precedes the timestamp of the previous All channels header. In simpler words, every new Tx timestamp should fit between the timestamp of the previous All channels header and a timestamp of the new All channels header. These rules are introduced to disable future-dating and back-dating of comments. If Bob breaks these rules, he can be disputed and lose his deposit.

Every Tx in the Channel tree must have the same channel id (CID) as the one in channel header. If Bob includes a Tx with a non-matching CID, he loses his deposit.



Channel tree

Channel header contains the following data:

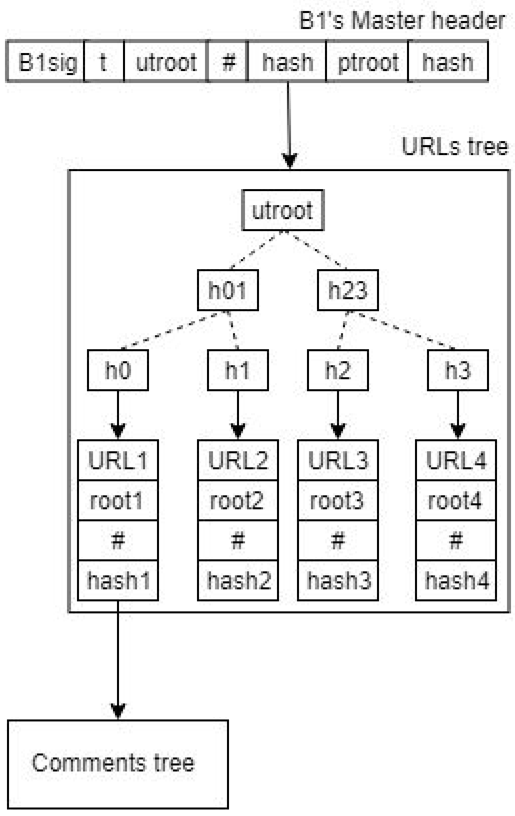
- **B1sig** - Bob1's signature. The signature must be valid.
- **CID** - a unique identifier for the state channel. This identifier must exist on the blockchain, referring to that specific channel.
- **root** - root element of the Merkle tree that contains all of the Txs that belong to the state channel. This root must match the root of the Channel tree.
- **#** - the number of Txs in the Merkle tree. The number must match the number of leaves in the Channel tree.
- **hash** - hash of the Merkle tree IPFS object.

Channel headers are objects used for channel closing, in case of no dispute.

Channel tree is susceptible to Merkle tree consistency proofs, thus enforcing the "add only" property to it.

4.6.4 URLs tree

URLs tree is used for grouping content by URL. It consists of Merkle tree and leaf objects acting like headers for Comments treel objects. URLs tree object's hash and root are present in the master header.



URLs tree

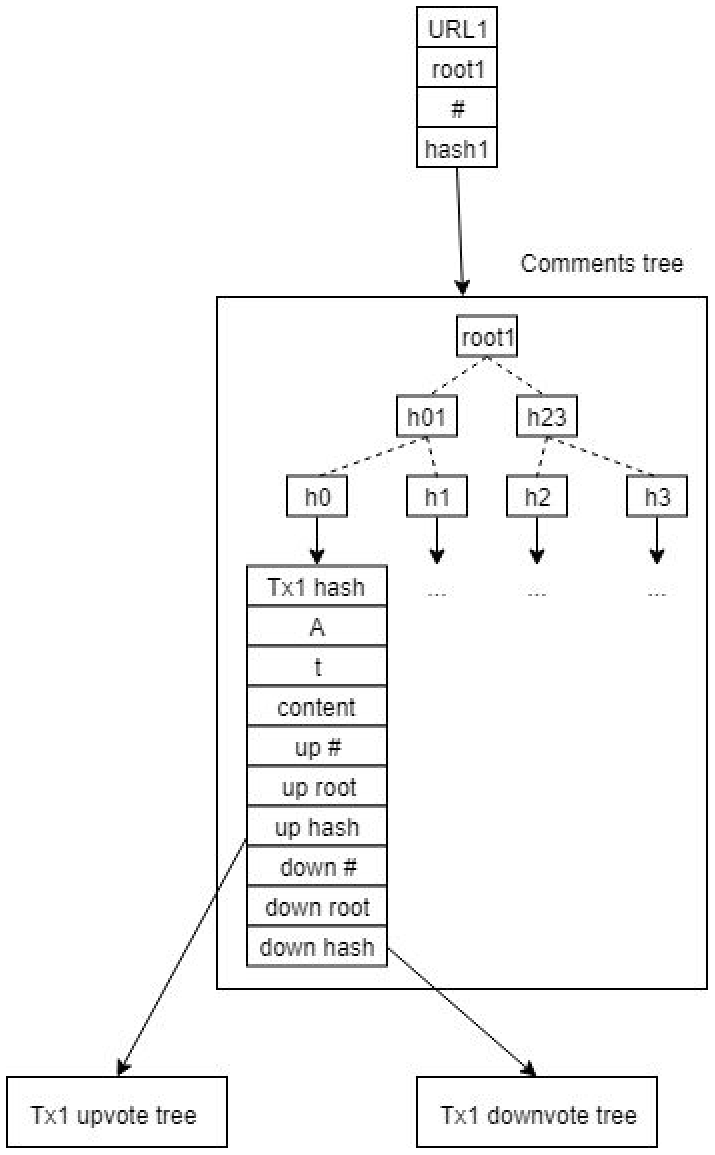
Leaf objects of URLs tree contain the following data:

- **URL1** - URL which comments refer to.
- **root1** - root element of the Url1 presentation object Merkle tree. This root must match the root of the Url1 Merkle tree.
- **#** - the number of elements in the Comments Merkle tree. This number must match the number of leaves in Comments Merkle tree.
- **hash** - hash of a Comments tree object.

Any attempt made by Bob to alter or delete the content of the Merkle tree, as well as to post content with a non-matching URL, can be detected by other participants in the system, and proven to the Smart contract.

4.6.5 Comments tree

Comments tree object contains all content associated with a specific URL. The object contains all comments posted on that URL, as well as the number of upvotes and downvotes for each comment. It consists of a Merkle tree and leaf objects representing a comment. Since it does not contain transactions, it is lightweight and used by readers who want to consume content for a specific URL.



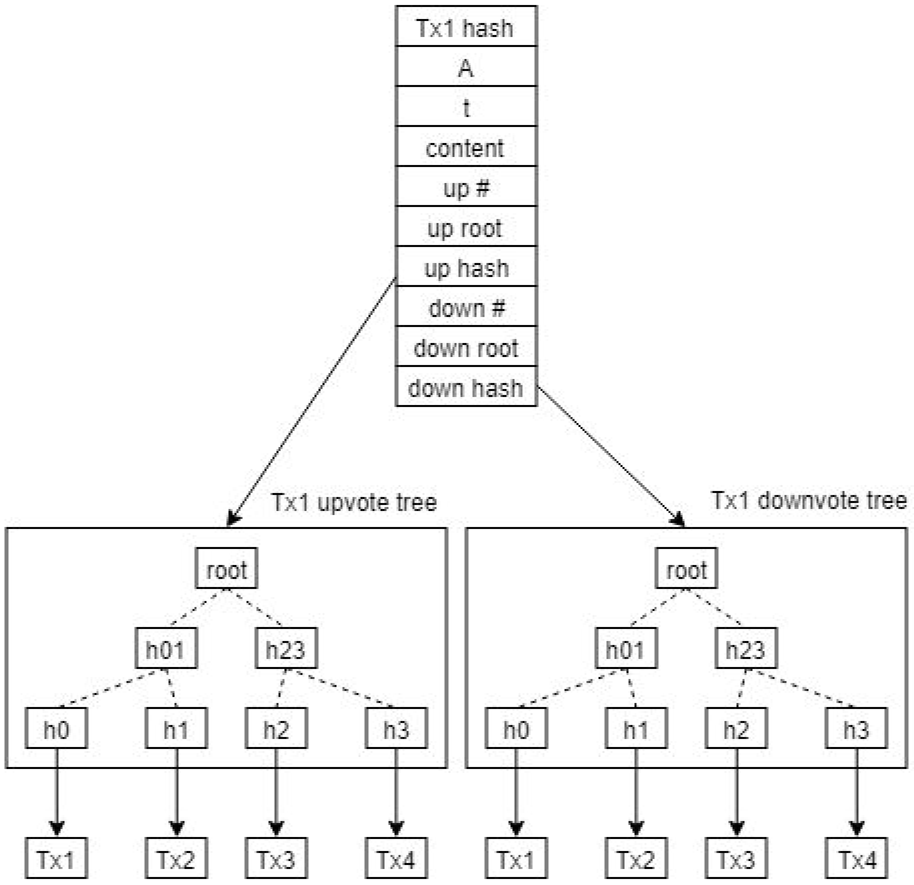
Comments tree

Leaf objects of Comments tree contain the following data:

- **Tx hash** - a hash of the comment transaction. Tx hash must point to a valid comment Tx that refers to that URL.
- **A** - Alice's address.
- **t** - the time of the comment (Ethereum block hash).
- **content** - content from comment transaction.
- **up root** - root element of the Upvote Merkle tree. This root must match the root in the Upvote tree IPFS object.
- **up #** - the number of elements in the upvote Merkle tree, and upvote count. This number must match the number of leaves in the Upvote tree IPFS object.
- **up hash** - hash of the upvote Merkle tree IPFS object.
- **down root** - root element of the downvote Merkle tree. This root must match the root in the Downvote tree IPFS object.

- **down #** - the number of elements in the downvote Merkle tree, and downvote count. This number must match the number of leaves in the Downvote tree IPFS object.
- **down hash** - hash of the downvote Merkle tree IPFS object.

4.6.6 Upvote/Downvote trees



Upvote/Downvote tree

Upvote/Downvote tree is used to group all of the upvote/downvote transactions that relate to the same comment transaction, in the form of a Merkle tree. Merkle tree leaves are pointing to upvote/downvote transactions made by any Alice to a specific comment. Again, Merkle tree hash and root is present in an object above - Comments tree object.

Upon adding a new element to the Merkle tree, Bob updates the comment object that the new element refers to, and updates the comments tree, URLs tree, and Master header accordingly.

Any attempt made by Bob to alter or delete the content of Merkle tree, as well as to add upvote/downvote Tx that points to the content Tx with non-matching URL, can be detected by other participants in the system, and be proven on the smart contract, resulting in Bob losing his deposit.

Upvote/Downvote trees are susceptible to Merkle tree consistency proofs, thus enforcing the "add only" property to them.

4.7 Setup

Bob has a large deposit in smart contract and repos on IPFS. Alice is about to open a state channel to Bob. Carol has submitted a one time fee to be registered as a system auditor and is subscribed to Bob's IPFS repos. All parties are monitoring the smart contract.

4.8 Opening a state channel

When Alice wishes to open a state channel to Bob, she constructs the transaction containing her deposit, and her signature, and sends the transaction to Bob. Upon receiving the channel opening transaction, Bob checks if the deposit and signature are valid. In case Bob wishes to proceed and open the channel with Alice, he co-signs the transaction and publishes the transaction to the smart contract. As soon as the transaction is mined, the state channel is considered to be open. After that, Bob and Alice can continue to exchange an array of transactions, each of which refers to comments publishing, upvoting or downvoting.

4.9 Transactions exchange protocol

After the state channel is open, Bob and Alice continue to exchange transactions respecting the following protocol:

1. Alice constructs a Tx containing a channel id, type, timestamp, counter, and a pointer to the content she wishes to publish (in case of a content transaction) or pointer to a Tx she wishes to upvote/downvote (in case of upvote/downvote transaction) and sends it privately to Bob.
2. Bob checks if all of the fields in the received Tx are valid. If he detects that there exists a problem, he responds to Alice addressing that problem, thus giving her a chance to correct the Tx. Otherwise, if the transaction is valid, Bob includes it into the Channel tree, constructs the new Channel header.
3. Bob then proceeds to update all of his data model structures, bottom up. First, he includes the new Channel header in his All channels tree and generates new All channels header. Then he includes that All channels header in his Providers tree. At the same time, he places the new transaction in the Comments tree object, updates the URLs tree accordingly, and links everything up in the Master header.
4. Alice checks if her transaction has been published. If so, the channel remains open, and she can proceed to publish another Tx. If Bob, for some reason, refuses to publish Alice's Tx, she can close the channel with Bob.

Note: Bob is allowed to batch transactions and do multiple updates at once, as long as he does not break the time restriction rules.

4.10 Closing a state channel

In the case of cooperative channel closing (no dispute), Alice or Bob use the Channel header to close the channel. If one party is trying to commit fraud and harm the other party, there is a dispute period to prove the fraud. Only Alice and Bob are permitted to close the channel in this way. After the dispute period has passed, other participants in the system can submit fraud proofs if any.

It is Bob's responsibility to make sure that the channel is closed with the last published Channel header. Closing the channel with a header that is not the last published one, is considered as spam from the system standpoint. Therefore, if Alice attempts to do so (to save money), Bob has a dispute period to present the latest valid header. If he fails to do so, Carol can present the last published valid header and collect a part of Bob's deposit, since Bob is considered as an accomplice in the fraud.

If Bob tries to close the channel with a header that contains a count higher than the actual number of published transactions, Alice can dispute him by requesting the proof for the transaction with the Channel Merkle tree leaf position matching the one from the header. Since she has not signed such a transaction, Bob is not able to provide the proof, and she gets to collect a part of Bob's deposit. Bob could try to add an already existing Tx to the Channel Merkle tree, but having duplicate Txs in the tree is also a cause for dispute.

4.11 Keeping the system in sync

Using the IPFS as the Tx exchange medium is not very practical as it would require a lot of IPFS objects downloads, which would degrade the system performance. For that reason, IPFS data model is used to verify the validity of the exchanged data, and as a backup option for data syncing if some of the Bobs become unresponsive. Instead, Bobs expose their data updates as a traditional server, by implementing a Pub-Sub engine. They would regularly publish their Tx deltas, accompanied with appropriate signed All channels header for validation purpose. Other Bobs would be subscribed to these deltas, and upon receiving them, they would apply them to their model. After applying them, if their generated header matches the received All channels header, they would include that All channels header into their model, and apply the deltas onto their presentational model.

If the generated header does not match the received one, that implies that there is a fault in the received data and a possibility for dispute. In that case, the "receiver Bob" would check the "server Bobs" IPFS model and commit a fraud proof to Smart contract if possible.

Also, Bobs would be required to serve content on-demand, for other Bobs that are lagging, for new Bobs that are syncing their model, and for Carols for auditing purpose.

As explained in chapter Providers tree, updates of Providers tree must be done regularly, meaning that the difference between a timestamp in the Master header and a timestamp in all of the included All channels headers must be less than a predefined time window. This restriction is introduced to prevent data back-dating. If any of the Bobs become unresponsive, other Bobs risk violating this restriction. In those situations, any of the remaining Bobs should issue a request for the missing unresponsive Bobs All channels header on the Smart contract. The unresponsive Bob then has a predefined period to expose the requested header to the Smart contract. If he fails to do so, he loses his deposit. To prevent other Bobs from draining a specific Bob's funds by continually requesting header updates on the Smart contract, even though that Bob is feeding them regular deltas, the request issuer must cover the cost of response transaction. Clients (Alices) are implemented in such a manner to abandon Bobs that are frequently unresponsive.

4.12 Serving the content

As a client wishes to access the content related to a specific URL, it would submit an HTTP request to any Bob (usually the one with the most recent Master header timestamp). Bob would then respond with the Comments tree object, a Master header, and a validity proof (Merkle tree audit paths, explained in Appendix A, leading all the way up to the Master header). After validating the received data, the client would assemble the data and present it to the reader.

It is important to note that at this point Bob is not aware if he is responding to a regular content consumer or a Carol, and must be aware that his response could be used for dispute purposes.

4.13 Disputes

A dispute is a process ignited when any of the Bobs is caught committing fraud. Fraud proof is a minimal set of data presented to the smart contract, enough to unambiguously prove that a fraud has been committed. Fraud proofs are part of a dispute transaction. Every fraud proof must contain at least one valid signature of disputed Bob. There are multiple ways Bob can be disputed, and therefore, there are multiple fraud proofs.

4.13.1 Dispute submission strategy

Any Carol that submits a fraud proof to the blockchain risks being copied by other Carols that have not done any of the validation work. Those copycat Carols can even submit a copied fraud proof Tx with a higher fee, to ensure that their Tx gets mined first. This kind of loophole favors the rich Carols over the honest ones.

Fraud proof submission is made by using a two-step commit-reveal scheme to prevent favoring copycat Carols. When Carol notices that Bob1 can be disputed, she constructs a dispute Tx (DTx) as well as proof of dispute Tx (PoDTx). DTx contains a fraud proof, while PoDTx contains a hash of DTx. In the first step (commit), Carol publishes the PoDTx to the blockchain, so she can prove that she had a DTx at a time of submission. This transaction is indistinguishable from any other transaction on the network, making it imperceptible to other dishonest copycat Carols. In step two (reveal), after PoDTx has been mined, Carol publishes the DTx to the blockchain. Once the DTx is mined, and if smart contract determines that fraud proof is valid and matches the one in PoDTx, Bob1 is considered as fraudulent, and his deposit and customers are lost.

Multiple Carols can submit disputes, both PoDTx and DTx. If fraud proofs are valid and PoDTx in the same block as first Carol, they are eligible for a share of Bob1's deposit. That way anyone who validated Bob1 gets a share. A list of Carols with valid proofs is now visible to anyone, where Carol1 is the first, Carol2 is second, et cetera.

Carol cannot submit multiple proofs since she needs to be registered. Registering in the system as an auditor requires a fee. This registration process prevents Carol to submit multiple proofs to increase her share of reward in the dispute process.

4.13.2 Dispute channels closing strategy

At this point, all opened channels to Bob1 need to be closed, and the last valid state (logical data model) need to be determined.

4.13.2.1 Round 1: Bobs submit their latest state

After a successful dispute, a consensus needs to be formed on what the state of all channels that are opened to Bob1 is. Bobs who want to participate in consensus can do so by submitting the latest state Tx (LSTx) containing their master header MH with timestamp t. Only those Bobs who assist in determining the consensus by submitting their MH are entitled to a share of disputed Bobs deposit.

4.13.2.2 Round 2: Carol1 validates the latest state Txs

After a period of LSTx submission, Carol1 first validates if all Bobs have submitted the latest state that is consistent with their model. If Carol can prove inconsistencies, she can claim new disputes against Bobs, just like when she noticed inconsistencies with Bob1. If a dispute is raised, disputed Bob's model is not used for consensus.

4.13.2.3 Round 3: Carol1 creates a list of channel closing headers

If there are no disputes for LSTx, she can start building a list of channel closing headers. Channel closing headers are Channel headers contained in the most recent valid All channels tree provided by the disputed Bob. Since all Bobs must update their model periodically, all models must contain precisely the same data in the past, while latest Txs may or may not be included yet. For every channel, she'll find the longest valid All channels Merkle tree in any model contained in LSTxs by applying the following algorithm:

- Each All channels header from committed LSTxs gets one weight point.
- Each All channels header that links to another, previous All channels header inherits the previous All channels header weight points.
- Channel headers from the All channels header with the highest weight get used for channels closing.
- If we have a tie situation (possible when Bob has been feeding one chain of data to some Bobs and the different chain to other Bobs), the All channels header with lesser hash wins.

Once Carol completes building a list for all channels, she includes it in a consensus transaction (CTx) and submits it to the blockchain.

4.13.2.4 Round 4: Channel closing and payout

If there are no disputes for CTx for a predefined period, each Carol in a list gets an equal share. However, if Carol2 can prove that Carol1 did not construct a valid header list in her CTx, she can dispute Carol1's CTx with her channel headers. Carol2 submits her sub-list of headers containing only disputed headers along with proofs of inclusion in selected LSTx. The sub-list of these committed headers are usually more recent ones that link back to the disputed ones. If no one disputes Carol2's claims, she can get Carol1's parts of share for every channel that is disputed. The process continues until all channels are closed.

When all channels are closed, Bobs and Carol that participated in dispute get paid. Also, all Bobs update their model according to the information provided in the channel closing headers.

4.13.3 Deposit distribution strategy

Disputed Bob's deposit is used to pay for transaction fees of all channel closing transactions that are not disputed. What is left from Bob's deposit is evenly divided to two halves: first half is divided equally among all Bobs who participated in dispute consensus, while the second half is divided among Carols who participated in the dispute process. The second half is divided as explained in the previous chapter.

The reason that other Bobs get half of the deposit (or what is left of it after channel closing) is to prevent the situation where a fraudulent Bob reports himself and collects his deposit.

4.13.4 Fraud proofs

Fraud proofs are part of a dispute transaction. All dispute functions are listed in the following chapters.

4.13.4.1 Master header

Malformatted header with valid signature - if a header contains a valid signature and any of its fields are in a wrong format, for dispute it is enough to submit that header.

dispute(masterHeader)

Two Master headers with # inconsistencies - if the Master header 1 is older than Master header 2, but count in Master header 1 is higher than the count in Master header 2, it is enough to submit the two headers as a proof.

dispute(masterHeader1, masterHeader2)

Two Master headers with the same timestamp - if there are two different Master headers with the same time, it is enough to submit the two headers as a proof.

dispute(masterHeader1, masterHeader2)

4.13.4.2 Providers tree

Providers tree disputes are all disputes that are related to Providers tree header (Master header), Providers Merkle tree, and objects that tree leaves are pointing to (BnACH).

Missing BnACH - if any All channels header (BnACH) is missing from the tree, Carol can ask Bob to submit a proof that missing BnACH is included in the Master header.

askForPath(masterHeader, Bn)

If Bob does not submit anything, and timeout expires, Carol wins the dispute. If Bob can produce proof, he does not lose his deposit, but this means that he constructed a valid tree but omitted a specific BnACH from it, and thus, this case becomes a candidate for a fraud claim.

Two ACH for the same Bob included - if there are two ACH entries for the same Bob, Carol should submit those two entries with proof of their inclusion in Providers tree.
dispute(masterHeader, path1, BACH1, path2, BACH2)

Time in BnACH bigger than in Master header - if BnACH with time t2 is included in the tree, but Master header contains t1, where t2 > t1
dispute(masterHeader, path, BnACH)

Malformatted BnACH header with valid signature - if any All channels header contains a valid signature and any of its fields are in a wrong format, Carol should submit that faulty All channels header (BnACH) and the proof that it is a part of a Master header.
dispute(masterHeader, path, BnACH)

Tree not ordered chronologically - if Providers tree is not ordered chronologically (by the time of Bob registration), Carol should submit a BnACH whose position in the tree does not match position in the smart contract.
dispute(masterHeader, path, BnACH)

Tree updated with the previous version of BnACH - if Merkle tree contains BnACH1 at time t1, and is updated with BnACH2 at time t2, where BnACH1 is newer than BnACH2 and t2 is higher than t1, Carol should provide proofs by presenting both All channels headers and proofs that they are included in Master headers.
dispute(masterHeader1, path1, BnACH2, masterHeader2, path2, BnACH1)

Tree updated with different BnACH with same time - if Merkle tree contains BnACH1 at time t1, and is updated with BnACH2 with the same time t1, where BnACH1 is different than BnACH2, Carol should provide proofs by presenting both All channels headers and proofs that they are included in Master headers.
dispute(masterHeader1, path1, BnACH2, masterHeader2, path2, BnACH1)

Empty tree object - if Providers tree is empty object Carol should submit a master header containing a hash to empty object.
dispute(masterHeader, treeObject)

Disputable BnACH is included - if All channels header (BnACH) that can be disputed exist in Merkle tree, Carol should submit the disputed All channels header, proof that it is included in the Master header, and the dispute transaction (DTx)
dispute(masterHeader, path, BnACH1, DTx)

Time in one of the headers lower than time window - if the difference between time in Master header and time in any of Bob's All channels headers (BnACH) is bigger than predefined time window, Carol should submit that All channels header and proof (audit path) that it belongs to the Master header.

dispute(masterHeader, path, BnACH)

4.13.4.3 All channels tree

All channel tree disputes are all disputes that are related to All channels tree header (BnACH), Merkle tree, and objects that tree leaves are pointing to (CH).

Missing Channel header - if any of the Channel headers are missing from the tree, Carol can ask Bob to submit a proof that missing CH is included in the All channels header.

askForHeaderAndPath(ACH, CH)

If Bob does not submit anything, and timeout expires, Carol wins the dispute. If Bob can produce proof, he does not lose his deposit, but this means that he did not provide a valid All channels tree and this case becomes a candidate for a fraud claim.

Two CH with the same CID included - if there are two CH entries with the same CID in the All channels tree, Carol should submit those two entries with proof of their inclusion in All channels tree.

dispute(ACH, path1, CH1, path2, CH2)

Malformatted Channel header with valid signature - if any Channel header contains a valid signature and any of its fields are in a wrong format, Carol should submit that faulty Channel header (CH) and the proof that it is a part of All channels header.

dispute(ACH, path, CH)

Tree not ordered chronologically - if All channels tree is not ordered chronologically (by the time of channel opening), Carol should submit a Channel header which position in the tree does not match the position in the smart contract.

dispute(masterHeader, path, CH)

Tree updated with the previous version of CH - if All channels tree at time 1 contains CH1, and is updated with CH2 at time t2, where t2 is higher than t1 and CH2 has a lower count than CH1, Carol should provide proofs by presenting both Channel headers and proofs that they are included in All channels headers.

dispute(ACH1, path1, CH1, ACH2, path2, CH2)

Two CH headers with same CID and # - if two different Channel headers with the same count for the same channel exist, Carol should submit both headers alongside proofs that they are included in All channels headers.

dispute(ACH1, path1, CH1, ACH2, path2, CH2)

4.13.4.4 Channel tree

Channel tree disputes are all disputes that are related to Channel tree header (channelHeader), Channel Merkle tree, and transactions that tree leaves are pointing to (Tx).

Missing Tx - if a Channel tree does not include Tx that is present in Comments tree or Upvote/downvote tree, that means that there is an inconsistency between the Presentational model and the Channel model. In this scenario, Carol should submit proof of the existence of that transaction in the Presentational model and ask Bob to submit proof of existence in the Channel model, with the same Master header. Since Bob is not capable of doing so, he loses his deposit.

askForHeaderAndPath(masterHeader, path, Tx)

Duplicate Tx - If a Channel tree contains two same Txs, it is enough for Carol to submit those two Txs with proof of their inclusion in the Channel tree.

dispute(channelHeader, path1, Tx, path2, Tx)

Malformatted Tx or content object - if a channel tree leaf points to a Tx containing a field that is in a wrong format, or is smaller or larger than the limit, Carol should submit the malformatted transaction or content and proof that it is included in the tree (audit path).

dispute(channelHeader, path, Tx, contentObject) or
dispute(channelHeader, path, Tx)

If Tx or contentObject contains fields that are too expensive to submit as proof, Carol can ask Bob to submit it.

askForTx(channelHeader, path) or
askForContentObject(channelHeader, path, Tx)

Missing URL - if a Channel tree does not include comment Tx with URL that is present in URLs tree, that means that there is an inconsistency between the Presentational model and the Channel model. In this scenario, Carol should submit proof of existence of that commentsHeader containing that URL in the Presentational model and ask Bob to submit proof of the existence in the Channel model, with the same Master header. Since Bob is not capable of doing so, he loses his deposit.

askForHeaderAndPath(masterHeader, path1, commentsHeader)

Tree not ordered chronologically - if a channel tree contains a Tx2 with time t2 before a Tx1 with time t1 both transactions and their audit paths should be submitted.

dispute(channelHeader, path1, Tx1, path2, Tx2)

Inconsistent tree - if a channel tree does not pass consistency proof validation.

dispute(channelHeader1, path1, channelHeader2, path2)

# smaller than leaf count - if # in the header is smaller than the number of leaves in Merkle tree, Carol should submit the last leaf and proof (audit path) that it is included in the tree.

dispute(masterHeader, path, lastLeaf)

# bigger than leaf count - if # in the header is bigger than the number of leaves in Merkle tree, Carol should ask for a transaction with count matching the one in the header.

askForLastLeaf(header)

If Bob cannot produce a valid answer, he loses the deposit. Otherwise, it means that he has a different header than Carol. That is now enough for Carol to a submit dispute with two different headers with the same #.

dispute(channelHeader1, channelHeader2)

Time in Tx bigger than in BnACH (Future-dated Tx) - if there is a time in Tx higher than the time in BnACH

dispute(BnACH, path, channelHeader, Tx, content)

Tx backdating - if there is a Tx with timestamp t, and All channels header with timestamp t1 (ACHt1), where t <=t 1, and All channels header with timestamp t2 (ACHt2) where t2> t1. If Tx is not a part of ACHt1 and is a part of ACHt2, Carol should submit the proof that that Tx is in ACHt2, and ask for proof of its inclusion in ACHt1. Since Tx is not in ACHt1, Bob is not able to present proof and loses his deposit.

askForProof(ACHt1, Tx, Txpath, CH, CHpath, ACHt2)

Wrong CID in Tx - if a Channel tree leaf points to a Tx with CID that does not belong to that Channel tree, Carol should submit proof of the existence of faulty transaction (Tx + audit path) in that specific CID header.

dispute(channelHeader, path, Tx)

# bigger than deposit - if the count in the header is bigger than the deposit in channel opening Tx it is enough to submit that header as a proof.

dispute(channelHeader)

Last header not used for channel closing - if a header other than the last is used for channel closing, the latest header should be submitted.

dispute(channelHeader)

4.13.4.5 URLs tree

URLs tree disputes are all disputes that are related to Master header (masterHeader), Merkle tree, and objects that tree leaves are pointing to (commentsHeader).

Missing URL - if a URL tree does not include URL that is present in Tx in Channel tree, that means that there is an inconsistency between the Presentational model and the Channel model. In this scenario, Carol should submit proof of the existence of that transaction in the Channel model and ask Bob to submit proof of existence in the Presentational model, with the same Master header. Since Bob is not capable of doing so, he loses his deposit.

askForPath(masterHeader, path, Tx, content)

Two Comments headers referring to the same URL - If there are two Comments headers, referring to the same URL in the URL tree, Carol should submit those two headers with proofs of their inclusion in URL tree.

dispute(masterHeader, path1, commentsHeader1, path2, commentsHeader2)

Malformatted Comments header - if a URLs tree leaf points to an object containing a field that is in a wrong format, Carol should submit the malformatted object and proof that it is included in the tree (audit path).

dispute(masterHeader, path, commentsHeader)

If the header contains fields that are too expensive to submit as proof, Carol can ask Bob to submit it.

askForObject(masterHeader, path)

Tree not ordered chronologically - all Comments headers must be sorted chronologically. Carol can submit two headers that break this rule.

dispute(masterHeader, path1, commentsHeader1, path2, commentsHeader2)

Tree updated with the previous version of Comments header - if Merkle tree contains commentsHeader2 at time t1, and is updated with commentsHeader1 at time t2, where commentsHeader1 is newer than commentsHeader2, and t2 is higher than t1, Carol should provide proofs by presenting both comments headers and proofs that they are included in Master headers.

dispute(masterHeader1, path1, commentsHeader2, masterHeader2, path2, commentsHeader1)

# smaller than leaf count - if # in Master header is smaller than the number of leaves in Merkle tree, Carol should submit the last leaf and proof (audit path) that it is included in the tree.

dispute(masterHeader, path, lastLeaf)

# bigger than leaf count - if # in Master header is bigger than the number of leaves in Merkle tree

askForLastLeaf(masterHeader1)

If Bob can produce a valid answer, it means that he has a different header than Carol. That is now enough for Carol to submit a dispute with two different headers with the same #

dispute(masterHeader1, masterHeader2)

Empty tree object - if Comments tree is empty object Carol should submit master header and a path to the empty object.

dispute(masterHeader, path, commentsTreeObject)

4.13.4.6 Comments tree

Comments tree disputes are all disputes that are related to Comments tree header (Comments header), Merkle tree, and objects that tree leaves are pointing to (Tx headers).

Missing Tx - if a comments tree does not include comment Tx that is present in a channel tree, Carol can ask Bob to submit same Master header with the path to the comments Tx
askForHeaderAndPath(masterHeader, path, Tx)

Two Tx headers referring to the same transaction - If there are two Tx headers with the same Tx hash field in the Comments tree, Carol should submit those two Tx headers, with proofs of their inclusion in the Comments tree.
dispute(masterHeader, path1, tx1Header, path2, tx2Header)

Malformatted tx header - if a tx header contains a field that is in a wrong format
dispute(masterHeader, path, txHeader)
If tx header contains fields that are too expensive to submit as proof, Carol can ask Bob to submit it.
askForObject(masterHeader, path)

Tree not ordered chronologically - if a channel tree contains a txHeader1 with time t2 before a txHeader1 with time t1 both headers and their audit paths should be submitted.
dispute(masterHeader, path1, txHeader1, path2, txHeader2)

Tree updated with the previous version of txHeader header - if Merkle tree contains txHeader2 at time t1, and is updated with txHeader1 at time t2, where commentsHeader1 is newer than commentsHeader2 and t2 is higher than t1, Carol should provide proofs by presenting both tx headers and proofs that they are included in Master headers.
dispute(masterHeader1, path1, txHeader2, masterHeader2, path2, txHeader1)

# in Comments header smaller than leaf count - if # in Comments header is smaller than the number of leaves in Merkle tree
dispute(commentsHeader, path, lastLeaf)

# in Comments header bigger than leaf count - if # in Comments header is bigger than the number of leaves in Merkle tree, Carol can ask Bob to submit a path to the last leaf in Comments tree using same Master header
askForLastLeaf(masterHeader)
If Bob can produce a valid answer, it means that he has a different Master header than Carol. That is now enough for Carol to submit a dispute with two different headers with same time
dispute(masterHeader1, masterHeader2)

Two headers with the same # - if two different transaction headers with the same upvote and downvote count exist
dispute(txHeader1, txHeader2)

Time in txHeader bigger than time in Master header - if txHeader with time t2 is included in the tree, but Master header contains t1, where t2 > t1
dispute(masterHeader, path, txHeader)

Not a comment Tx - if Tx with the wrong type is included in the Merkle tree
dispute(masterHeader, path, Tx)

Non-matching A, time, content - if a tx header contains a field that does not match a field in Tx included in the channel tree
dispute(masterHeader, path1, txHeader, path2, Tx, content)

Non-matching URL - if a tx header is included in Comments tree for URL1, but Tx hash in Tx header is pointing to a Tx for URL2
dispute(masterHeader, path1, txHeader, path2, Tx)

4.13.4.7 Upvote/Downvote tree

Upvote/Downvote tree disputes are all disputes that are related to Upvote/Downvote tree header (txHeader), Merkle tree, and upvote/downvote transactions that tree leaves are pointing to (Tx).

Missing Tx - if an Upvote/downvote tree does not include upvote/downvote Tx that is present in channel tree, Carol can ask Bob to submit same Master header with the path to Tx
askForHeaderAndPath(masterHeader, path1, Tx)

Duplicate Tx - If an Upvote/downvote tree contains two same Txs, it is enough for Carol to submit those two Txs with proof of their inclusion in the Upvote/downvote tree.
dispute(masterHeader, path1, Tx, path2, Tx)

Malformatted Tx - if an Upvote/Downvote tree leaf points to a Tx containing a field that is in a wrong format, or is smaller or larger than the limit, Carol should submit the malformatted transaction or content and proof that it is included in the tree (audit path).
dispute(masterHeader, path, Tx)
If Tx contains fields that are too expensive to submit as proof, Carol can ask Bob to submit it.
askForTx(masterHeader, path)

Txs not ordered chronologically - if an upvote/downvote tree contains a Tx2 with time t2 before a Tx1 with time t1
dispute(masterHeader, path1, Tx1, path2, Tx2)

Inconsistent tree - if a upvote/downvote tree does not pass consistency proof validation
dispute(masterHeader1, path1, masterHeader2, path2)

# smaller than leaf count - if # in header is smaller than number of leaves in Merkle tree, Carol can submit the last leaf
dispute(masterHeader, path, lastLeaf)

# bigger than leaf count - if # in the header is bigger than the number of leaves in Merkle tree
askForLastLeaf(masterHeader1)
If Bob can produce a valid answer, it means that he has a different header than Carol. That is now enough for Carol to submit a dispute with two different headers with the same #
dispute(masterHeader1, masterHeader2)

Time in tx bigger than time in txHeader - if tx with time t2 is included in the tree, but txHeader contains t1, where t2 > t1
dispute(masterHeader, path, tx)

Not upvote/downvote Tx - if Tx with the wrong type is included in the Merkle tree
dispute(masterHeader, path, Tx)

Upvote/Downvote Tx points to a wrong comment Tx - if upvote/downvote Tx points to comment Tx that is not present in the header of the Merkle tree (comment Tx hashes in txHeader and upvote/downvote Tx do not match)
dispute(masterHeader, path, Tx)

Tx back-dating - if an upvote/downvote tree at time t1 does not contain Tx1 with time t1, but at time t2 contains Tx1 with time t1
askForPath(masterHeader1, masterHeader2, path2, Tx1)

4.14 Fraud claims

Unlike fraud proofs, fraud claims refer to fraudulent situations that cannot be proven to the smart contract, usually due to the size of the proof. For example, if the root in any of the signed headers does not match the root in the linked IPFS Merkle tree, to prove this fraud, it would be necessary to submit the entire Merkle tree. Other examples include situations where Bob deliberately omits an element in the IPFS Merkle tree (or even a whole sub-tree for that matter).

However, since all of the data is public, any fraudulent behavior is visible and easily verifiable by every participant in the system. As all Bobs are obligated to verify the validity of the data received from other Bobs, they can quickly detect this kind of situation. Upon doing so, they are not to include the fraudulent All channels header into their data model. If they include the fraudulent All channels header into their model, they are treated the same way as the Bob who published the fraudulent header. Instead, they should file a fraud claim, specifying the invalid data, to the smart contract. There are two possible outcomes:

- If the fraud claim turns out to be valid, the fraudulent Bob does not lose his deposit (since the claim cannot be proven) but gets "excommunicated" from the system, as no other Bob should include any of his future published data on account of the fraud claim. Exclusion from the system causes him to lose all of his clients, as their future posts and actions will not be a part of the overall system data model. The clients are implemented in such a manner that after verifying the fraud claim, they start closing channels with dishonest Bob with the last valid state.
- If the fraud claim turns out to be invalid (attack on Bob), it gets ignored, and Bob who posted the claim gets expelled from the future system functioning. If the fraud claim is posted by Carol, and it turns out to be invalid, it is just ignored.

Once a valid fraud claim is mined, other Bobs should use the fraud claim transaction in their Providers tree, instead of the All channels header of the Bob who committed the fraud.

4.15 The data withholding problem

At any point of the system functioning, any of the Bobs can go rogue, stop publishing data using Pub-Sub engine, and stop publishing IPFS objects. After being pinged on the blockchain, they could publish a header that contains a pointer (hash) to an unpublished IPFS object. In this situation other Bobs cannot file a fraud claim, as rogue Bob could, at any time, reveal the missing data and disprove them, causing them to lose business.

4.15.1 The solution

If Bob decides not to publish any piece of information, instead of including his All channels header to a Providers tree, other Bobs are obligated to include a "missing data" statement, indicating the exact piece of information that is missing. There is a certain time window for Bob to restore the missing data, as he could be facing server downtime. If he fails to do so in that predefined time window, the majority of honest Bobs will have a "missing data" statement in their Providers tree (instead of Bobs All channels header), and Bobs latest valid All channels header. In the situation where the time window has elapsed, and the majority of Bobs do have a "missing data" statement, one of the Bobs collects those statements and submits them to the blockchain together with proofs (audit paths, explained in Appendix A) to other Bobs Providers headers.

Please note that a simple majority of Bobs with "missing data" statement is not good enough in this scenario, as a dishonest Bob can spawn a network of his loyal Bobs (an operation that would be highly costly to him, but one that must be taken into consideration) and feed the data only to them. Therefore, an algorithm should be used that takes into account other Bob’s overall reputation, number of successfully closed channels, number of open channels and overall track record.

Once the proof is submitted to the blockchain, a channel closing procedure described in the chapter Dispute channels closing strategy commences. The only difference is that other Bobs do not have to post the latest valid state, as it was already gathered together with the "missing data" statement. The channels are closed using the dishonest Bobs deposit, but in this case, he does not lose all of his deposit, as we cannot be sure that he was withholding data deliberately. The aftermath is that he loses a part of his deposit required for channels closing, and it is up to his customers to decide if they want to reestablish collaboration with him or migrate to another Bob.

If a certain Bob publishes a “missing data” claim for another Bob that is, in fact, providing data, clients (Alices) are written to abandon the Bob that posted the false “missing data” claim.

4.16 Caveats

4.16.1 Fraud claims versus disputes

There is a disproportion in repercussions that Bob faces when there is a dispute filed against him, compared to the situation where a fraud claim is filed against him. In both situations, he is committing fraud, but the first one causes him to lose the deposit, and the second one, since it cannot be proven to the smart contract, only makes him lose business.

We are working on the protocol that should level the playing field for these two situations.

4.16.2 Performance

If any of the Bobs decide to shut the servers down and only update the IPFS objects, there would be no way to hold him responsible, but he would degrade the overall performance of the system since data gathering, and verification would require multiple IPFS objects download. A good thing about this scenario is that, once any of the other Bobs has processed unresponsive Bob’s data, he could continue to feed them to the rest of the system.

Another issue is related to the fact that all of the Bobs should keep track of all of the system state. This could lead up to the system bloating over time, and as a consequence, increase the price of the comments fees. We are working on the protocol that should allow Bobs to commit the validated parts of the data model on the blockchain, and only keep track of the updates, upon doing so. The overall system state would then be constructed by merging the parts committed on the blockchain and the "live" updates.

4.16.3 Harmful content

Balancing between providing a censorship-resistant platform, and protecting consumers from harmful content is not an easy task. POC version of the platform will not include this kind of protection, but later, commercial versions should focus on providing machine learning based, content filtering plugins that can be configured and executed on the client side, based on the client preferences.

4.16.4 Block reordering

Since block hashes are used as timestamps in system functioning, block reordering could impact this functionality. This impact should be a subject of a thorough analysis.

5 Future functionalities

This chapter contains future functionality, to be implemented after the TFA layer 2 protocol is completed.

System optimization

System optimization refers to performance improvements of the layer 2.

Contract stabilization and optimization

Implies security auditing of the smart contracts, running them against ConsenSys guidelines, and possibly migration from Solidity/Truffle to Viper/Populus if the latter turns out to be safer and more practical.

Conversation threads

Implementation of functionality that allows responding to comments.

Donations for quality content

Donations for quality content implies an implementation that provides an easy way of transferring funds from a satisfied reader to the content creator.

Explore comments option

Layer 2 implementation would only provide an ability to browse comments associated with a visited URL. To make the product more user-friendly, and give the users insight into the domains with most activity, we will implement various browse options, such as browsing by a specific domain (not URL), by recent activity, by the author of the comments, by trending topics, et cetera.

Author profile page

Implementation of a profile page with provided author info, and all of his comments.

Subscription to authors of interest

Implementation of possibility to subscribe to a particular author of interest and get notified when he posts something new to the system.

Content filtering strategy

Dealing with sensitive or harmful content in a censorship-free system is a delicate task that should be approached thoroughly. The first iteration could rely on identity systems such as uPort, for our users’ registration. Ultimately content filtering policy should be defined on the client side, based on users preferences, with open filter plugins that rely on machine learning.

Create a website

Implementation of a native website that binds all of the functionalities of the extension, allows browsing by various categories, implements user profiles, allows easy auditor and provider registration, et cetera.

Mozilla addon

Implementation of a Mozilla add-on, aside from the Chrome extension.

Content bounties

Implementation of a system where a group of users, interested in a particular users opinion on some topic, could offer a bounty that could be collected by the user of interest, upon providing comment on the requested subject.

Content challenging system

Integration with market prediction protocols such as Augur, to challenge other users content validity or truthfulness.

Android app

Implementation of an Android app.

iOS App

Implementation of an iOS app.

Integration with BAT

Integration with BAT to provide our users a share of profit for content consumption.

Integration with identity systems

Integration with identity systems such as uPort for user registration.

Integration with MetaMask

Integration with MetaMask to simplify the payment process.

Integration with native portals

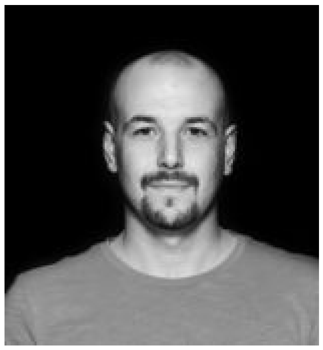
Implementation of an API that would allow any portal to use TFA platform as their commenting system.

6 ICO and Funds Distribution

TBD if we decide to go with an ICO.

7 Team

Bojan Jovin (CEO)



Has an MSc degree in Electrical and Computer Engineering - Computer Science, University of Novi Sad, Serbia.

Work experience:

- Software developer and project manager for Schneider Electric DMSNS (Serbia, 2009-2015)
- Java developer for Ixaris Systems Ltd. (Malta, 2015-2016)
- Senior Java developer and consultant for Codecentric (Serbia, 2016-2018)
- CEO/Co-founder in Merkle Blue DOO (Serbia, 2018-present)

Successfully carried out a number of large international projects as a project manager, five of which in China, one in Argentina, Mexico, and England.

Oracle Certified Professional, Java SE 8 Programmer.

Experienced in Javascript language and Javascript frameworks (Nodejs, Npm, Angular, React, Redux, Bootstrap, et cetera).

Experienced in agile software development methodologies (carried out a role of SCRUM master).

Familiar with software architecture and development paradigms such as DOA, Event Sourcing, Microservices, TDD, Clean Architecture, SOLID, et cetera.

Extremely interested in blockchain ecosystem since 2013. Has experience working with Solidity and Truffle framework.

Fluent in English.

Nebojsa Konstantinovic (CTO)

Has an MSc degree in Electrical and Computer Engineering - Computer Science, University of Novi Sad, Serbia.

Work experience:

- Junior software developer for Schneider Electric DMSNS (Serbia, 2009-2010)
- Self-employed at Scraperware Novi Sad (Serbia, 2012-2018)
- CTO/Co-founder in Merkle Blue DOO (Serbia, 2018-present)

Worked on a series of small web projects involving Web design, HTML/CSS/JS, PHP, SEO.

Experienced Java developer, worked with Vaadin framework, Spring framework, Selenium, TDD.

Experienced in Javascript language and Javascript frameworks (Nodejs, Npm, React, Redux, MaterialUI, et cetera).

Experienced in agile software development methodologies.

Experienced in development of Ethereum smart contracts. Has experience working with Solidity and Truffle framework.

Extremely interested in blockchain ecosystem since 2011.

Fluent in English.

8 Glossary

phrase	meaning
TFA	The First Amendment
IPFS	Inter-Planetary File System
persona	user profile
IP	Internet Protocol
EVM	Ethereum Virtual Machine
Tx	Transaction
web3	Ethereum javascript API
UI	User Interface
Ganache	Local testnet client
Ropsten	Public Ethereum testnet
HTTP	HyperText Transfer Protocol
CID	Channel Identifier
Bsig	Bob’s signature
t	timestamp (block hash)
utroot	The root element of the URLs tree
uthash	IPFS hash for the URLs tree object
ptroot	The root element of the Providers tree
pthash	IPFS hash for the Providers tree object
up root	The root element of the Upvote tree
down root	The root element of the Downvote tree

MH	Master Header
ACH	All Channels Header
BnACH	All Channels Header for n-th Bob
CH	Channel Header
DTx	Dispute Transaction
PoDTx	Proof of Dispute Transaction
LSTx	Last State Transaction
CTx	Consensus Transaction
BAT	Basic Attention Token
uPort	Identity system
Augur	Market prediction platform
ICO	Initial Coin Offering
POC	Proof Of Concept

9 Appendix A - Merkle tree audit paths

A Merkle audit path for a leaf in a Merkle Hash Tree is the shortest list of additional nodes in the Merkle Tree required to compute the Merkle Tree Hash for that tree. Each node in the tree is either a leaf node or is computed from the two nodes immediately below it (i.e., towards the leaves). At each step up the tree (towards the root), a node from the audit path is combined with the node computed so far. In other words, the audit path consists of the list of missing nodes required to compute the nodes leading from a leaf to the root of the tree. If the root computed from the audit path matches the true root, then the audit path is proof that the leaf exists in the tree.

Given an ordered list of n inputs to the tree, D[n] = {d(0), ..., d(n-1)}, the Merkle audit path PATH(m, D[n]) for the (m+1)th input d(m), 0 <= m < n, is defined as follows:

The path for the single leaf in a tree with a one-element input list D[1] = {d(0)} is empty:

PATH(0, {d(0)}) = {}

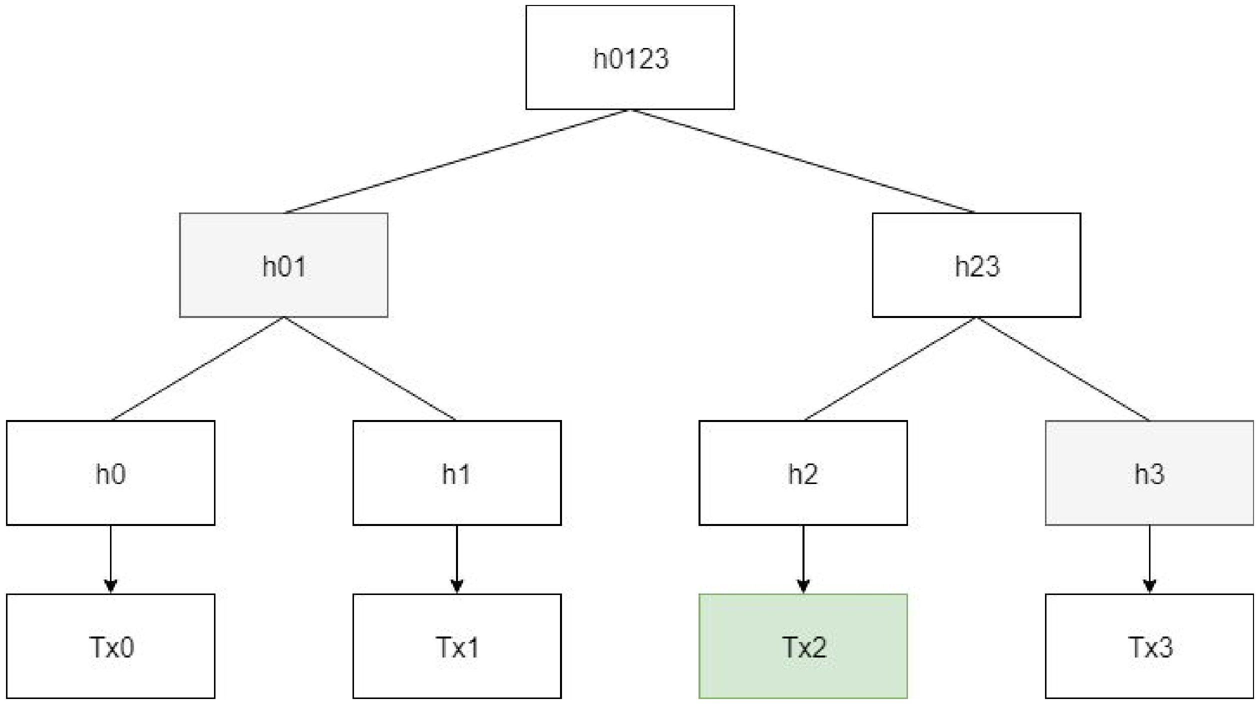
For n > 1, let k be the largest power of two smaller than n. The path for the (m+1)th element d(m) in a list of n > m elements is then defined recursively as

PATH(m, D[n]) = PATH(m, D[0:k]) : MTH(D[k:n]) for m < k; and

PATH(m, D[n]) = PATH(m - k, D[k:n]) : MTH(D[0:k]) for m >= k,

where : is concatenation of lists and D[k1:k2] denotes the length (k2 - k1) list {d(k1), d(k1+1),..., d(k2-1)} as before.

In practice, this means that for the Tx2 element on the picture below, an audit path would be h3 and h01, as those are the minimum subset of Merkle tree elements required to compute its root.



Merkle tree audit path

[RFC 6962]

10 Appendix B - Merkle tree consistency proofs

Merkle consistency proofs prove the append-only property of the tree. A Merkle consistency proof for a Merkle Tree Hash MTH(D[n]) and a previously advertised hash MTH(D[0:m]) of the first m leaves, m <=n, is the list of nodes in the Merkle Tree required to verify that the first m inputs D[0:m] are equal in both trees. Thus, a consistency proof must contain a set of intermediate nodes (i.e., commitments to inputs) sufficient to verify MTH(D[n]), such that (a subset of) the same nodes can be used to verify MTH(D[0:m]). We define an algorithm that outputs the (unique) minimal consistency proof.

Given an ordered list of n inputs to the tree, D[n] = {d(0), ..., d(n-1)}, the Merkle consistency proof PROOF(m, D[n]) for a previous Merkle Tree Hash MTH(D[0:m]), 0 < m < n, is defined as:

PROOF(m, D[n]) = SUBPROOF(m, D[n], true)

The subproof for m = n is empty if m is the value for which PROOF was originally requested (meaning that the subtree Merkle Tree Hash MTH(D[0:m]) is known):

SUBPROOF(m, D[m], true) = {}

The subproof for m = n is the Merkle Tree Hash committing inputs D[0:m]; otherwise:

SUBPROOF(m, D[m], false) = {MTH(D[m])}

For m < n, let k be the largest power of two smaller than n. The subproof is then defined recursively.

If m <= k, the right subtree entries D[k:n] only exist in the current tree. We prove that the left subtree entries D[0:k] are consistent and add a commitment to D[k:n]:

SUBPROOF(m, D[n], b) = SUBPROOF(m, D[0:k], b) : MTH(D[k:n])

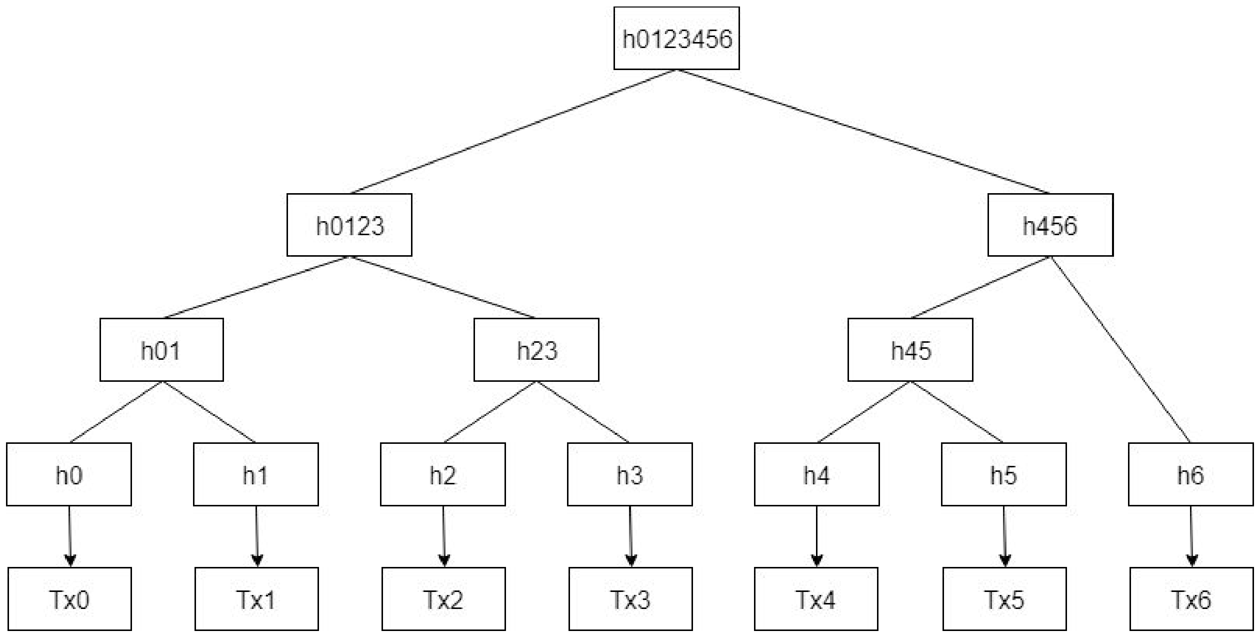
If m > k, the left subtree entries D[0:k] are identical in both trees. We prove that the right subtree entries D[k:n] are consistent and add a commitment to D[0:k].

SUBPROOF(m, D[n], b) = SUBPROOF(m - k, D[k:n], false) : MTH(D[0:k])

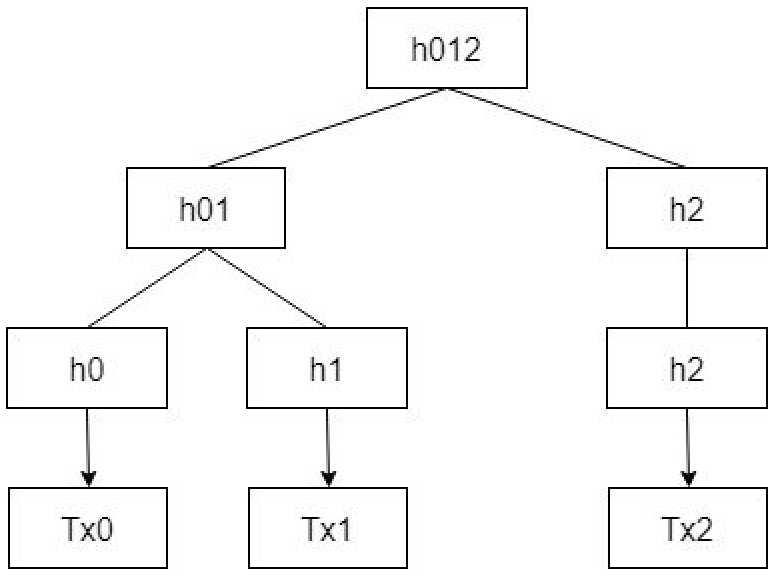
Here, : is a concatenation of lists, and D[k1:k2] denotes the length (k2 - k1) list {d(k1), d(k1+1),..., d(k2-1)} as before.

The number of nodes in the resulting proof is bounded above by ceil(log2(n)) + 1.

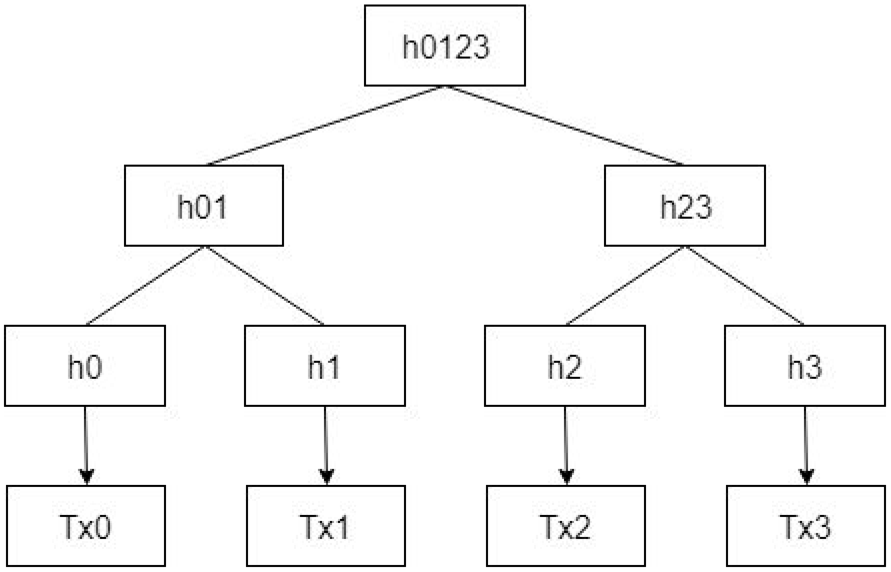
For example, if we have a Merkle tree with 7 leaves:



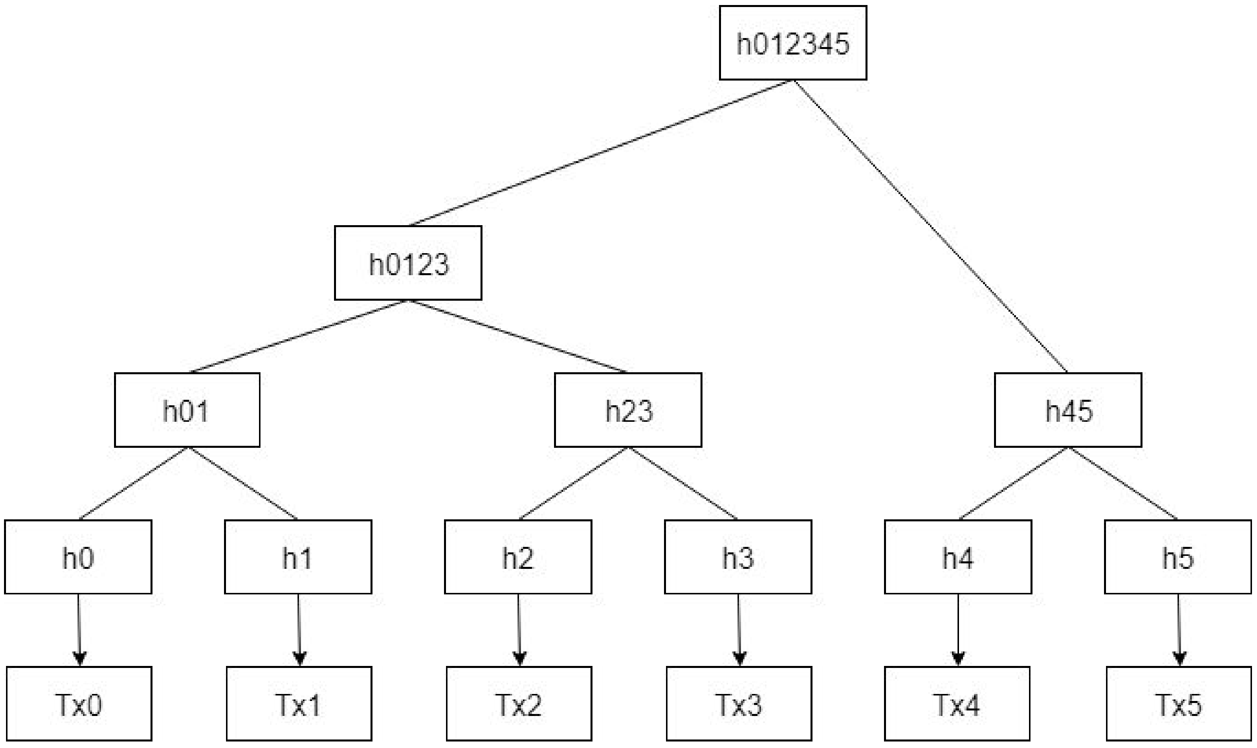
The audit path for Tx0 is [h1, h23, h456]
The audit path for Tx3 is [h2, h01, h456]
The audit path for Tx4 is [h5, h6, h0123]
The audit path for Tx6 is [h45, h0123]
The same tree, built incrementally in four steps:
Step1: 3 elements



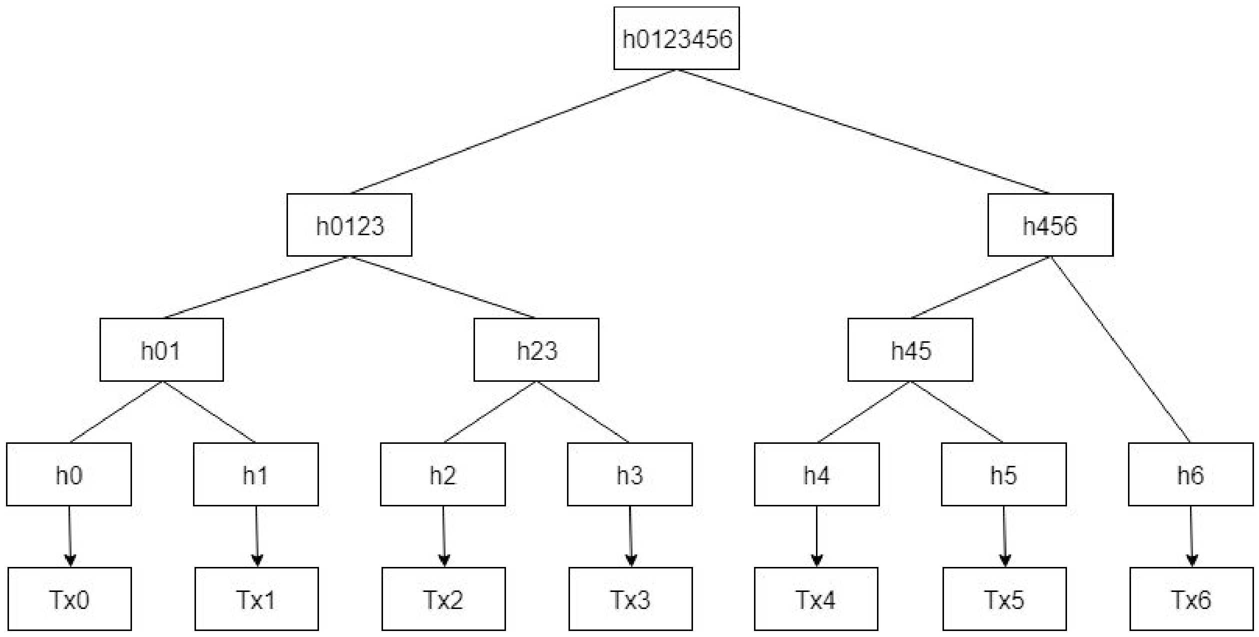
Step 2: 4 elements



Step 3: 6 elements



Step 4: 7 elements



The consistency proof between the first tree and last tree is PROOF(3, D[7]) = [h2, h3, h01, h456].

The consistency proof between the second tree and last tree is PROOF(4, D[7]) = [h456].

The consistency proof between the third tree and last tree is PROOF(6, D[7]) = [h45, h6, h0123].

[RFC 6962]