catalyst

Catalyst Network

Technical White Paper

Catalyst Network - Technical White Paper (DRAFT - V1)

James Kirby*[1] and Joseph Kearney†[2]

[1]Catalyst Foundation
[2]AtlasCity

December 6, 2019

Abstract

The Web3 movement has become a war cry for a new and better web. With the rise in popularity in consensus protocols, where users demand more control over the finances, we are now demanding more privacy, more security and more control over our data. Web3 is the idea that the same stateful protocols that are revolutionising the financial industry can also revolutionise the much-broken web. The Catalyst Network aims to be a full stack solution for developers and business to build feature rich applications for the decentralized web.

This document is a work in progress and is subject to change.

*nshcore@protonmail.com
†joseph.kearney@atlascity.io

Contents

Introduction

With the creation of Bitcoin [1], we witnessed the opening of Pandora's box in terms of financial ingenuity, self-sovereignty and a shift of power from the financial elite to the people. Since the 1980's people like David Chaum have theorised about the ideas of distributed consensus-based protocols [2]. Although a technical and social breakthrough, Bitcoin now comparatively is like sending money via cheques in terms of the raw throughput and functionality.
With the creation of Ethereum [3], we witnessed the opening of Pandora's box for a second time. We were introduced to the concept of the world state and distributed computing which until its creation had only been theorized by people like Nick Szabo 20 years ago. The idea of decentralised finance and peer-based crowdfunding further challenged the grasp of the financial world. At this time new societal concepts such as Web3 were introduced.
Consensus-based protocols like blockchains introduced a method for each participant in a network to hold and transfer state in a digitally native format, applying stateful protocols to the traditionally stateless protocols that power the web, gives the web a native mechanism to authoritatively say who owns what, and who has permissions to do an action.
But as groundbreaking a technology Etheruem is, it has some major problems in terms of scalability of feature-rich functionality. More generally with the current Web3 paradigm, trying to build decentralised applications often forces developers to adopt multiple protocols, multiple native tokens, and multiple technology stacks in return for simplistic functionality compared to our stateless ancestor protocols.
Our approach for building Catalyst was to solve these core issues, and to:

- Build a true Byzantine fault-tolerant consensus that becomes increasing scalable, decentralised and secure at scale.
- Provide a feature-rich full-stack solution to developers who want to embrace the web3 ethos.
- Rethink the economic incentives around blockchain systems, to be fairer to all participants.

1 Catalyst Consensus Mechanism

Distributed networks have no single point of trust to determine the validity of transactions, so consistency must be ensured by other methods. Typically this requires a majority of the network's participants to agree on a particular update of the ledger and the changes to account balances held on the ledger. Blockchain technologies generally employ Proof-of-Work (PoW) and occasionally Proof-of-Stake (PoS) mechanisms in order to gain consensus across a network. PoW originates from hashcash [4], which was designed as a prevention mechanism for email spam. This idea was then developed for the bitcoin blockchain, it required miners to perform a computationally hard problem in order for them to expend energy and there by resources. This dissuades mining nodes from acting maliciously, as both the transactions they have added as well as the work they have performed can be verified. If their work is poor then the block they are trying to append to the blockchain will be rejected *i.e.* will not be considered the longest chain. This technique for consensus is Byzantine fault tolerant and can not be understated as the critical factor that has allowed blockchain technologies to grow. However, this has two critical flaws, firstly the amount of expended and wasted energy is extraordinary especially on large networks where the potential rewards are large. Secondly there has been a general trend towards centralisation at scale, as mining pools hold increase their share of the network power (and thereby the networks rewards). Other networks employ a small amount of trusted nodes that ensure the validity of transactions, but this is also highly centralised and almost as fallible as the single point of failure systems that DLTs endeavors to avoid.

Catalyst integrates a newly designed consensus mechanism, based on Probabilistic Byzantine Fault Tolerance (PBFT). However, Catalyst gains consensus through a collaborative and fair mechanism by voting rather than through a competitive process, meaning that all honest work performed by nodes on the network benefits the security of the network and that all successful participating nodes are rewarded equally. The consensus for Catalyst can therefore be more accurately described as a PBFT voting mechanism. For each ledger cycle a random selection of worker nodes are selected, the nodes become the producers for a cycle or number of cycles. The producer nodes perform work in the form of compiling and validating transaction thereby extracting a ledger state change for that cycle.

The protocol is split into four distinct phases:

- Construction Phase - Producer nodes that have been selected create what they believe to be the correct update of the ledger. They then distribute this proposed ledger update in the form of a hash digest.
- Campaigning Phase - Producer nodes designate and declare what they propose to be the ledger state update as determined by the values collected from other producer nodes in the producer pool.
- Voting Phase - Producer nodes vote for what they believe to be the most popular ledger state update as determined by the producer pool according to the candidates for the most popular update from the other producer nodes.
- Synchronisation Phase - The producers who have computed the correct ledger update can broadcast this update to the rest of the network.

This section is based on the work set out in [5], where the original research into the creation of a new consensus mechanism is laid out. The various parameters and thresholds mentioned in this chapter and their impact on the levels of security and confidence in the successful production of a ledger state update are discussed in the following paper [6].

1.1 Producer Node Selection

Original work for the Catalyst consensus mechanism states that the peers are selected with relation to the persistent identifier and the hash of the previous data. But as the persistent identifier

can be manipulated by a user and thereby weighted in their favor of selection to become a producer, this is not usable.

Research into a randomly decided decentralised autonomous organisation (RANDAO) provides a viable alternative [7][8]. This is a process by which each user creates their own random value and by combining these random numbers across the network they calculate a global pseudo-random number. The larger the network the more random the number will be. The proposed process works as follows:

- Define p as the maximum integer that the hashing function used can represent.
- Each node in the worker pool generates a random number r in the range [0, p].
- To this random number the hash of the previous ledger state, D, must be added mod p.
- Each node then creates a Blake-2b hash of the combined random number $H(r + D)$.
- Each node must then send their value r to the contract.
- If they do not send their r value they are not eligible to become a producer node.
- Each node in the worker pool sends their $H(r + D)$ to the global state. This creates the global random value G. This is done through addition of all the random numbers mod p.
- The global state must determine:
  - Whether a particular node did submit all correct information, *i.e.* r, and $H(r + D)$.
  - That a particular node did in fact use the D value when generating the random number. This is done by taking the r value submitted by the user and hashing with D.
  - Ensuring the node has paid a sufficient stake to take part in the selection process.
  - Validating that each worker has only distributed one random number to the smart contract.
- Failure of any one of these points means the smart contract will not accept the submission of a random number from r and and the created stake will be lost.
- This global random number can then be used to determine the producers for the next cycle(s).
- This is done by determining the nodes that have a $H(r + D)$ closest to the G value using a XOR Hamming distance search on the binary tree.

This method is secure from manipulation because hashing algorithms are one way functions, meaning that there is no provably efficient method to inverse a hashing function, *i.e.* retrieving a message m from a digest $H(m)$. If a node does not input a value into the global state then they are not eligible for selection for becoming a producer for that cycle(s).

Inclusion of the D value is necessary as this will prevent a producer from using known random values to create a desired digest that gives them an advantage when being selected. The value for D must fulfill two rules, firstly it must always be the same for all nodes in the worker pool, secondly it must change with each draw of a random number when determining the random selection of producer nodes. This prevents a user creating a hash using known input and digest combinations in order to gain an advantage when being selected. Furthermore, if the hash of the previous ledger state is used as D it ensures that a prospective producer node knows the current ledger state. If they do not then their random number will be invalid as $H(r+D_{prod}) \neq H(r+D)$.

As described in RANDAO [8], this can be further extended to implement staking. Nominal contribution fees can be added in order to prevent DDOS and Sybil attacks without disincentivising participation. There is also the additional benefit of simplification of the overall consensus mechanism as it removes the need for a queuing mechanism, as well as producing a verifiable method of keeping track of which nodes are registering to be workers for any given cycle. It also in turn provides evidence to other nodes on the network who the producers for any given cycle

are as the process will be verifiable.

This scheme will provide the Catalyst consensus mechanism with a verifiable, reliable and secure mechanism to generate a pool of producers. By randomly assigning which workers get to participate in the next ledger cycle, fairness is ensured.

1.2 Notation

For the sake of clarity, here is a dictionary of the variables used in the proceeding description of the consensus mechanism:

- p - A single producer that has been selected randomly to perform work for the network for a given cycle.
- P - The pool of producers who have been selected through the producer selection mechanism for the particular cycle being described. It is made up of a set of producers p.z
- T - The mempool of any given producer p. It contains a group of transactions that will be used in the update for a given cycle.
- t - A single transaction that is found within the mempool T.
- u - The local ledger state update generated by a single producer node. This is what that particular node believes to be the correct update for the given cycle.
- σ - A pseudorandomly generated number, used as a salt.
- E - The list of entries found within a single transaction t. A producer will produce one of these lists for each valid transaction in T.
- d - A hash tree containing all the signatures extracted from the transactions in T.
- H - A hash of each list E combined with the salt σ.
- L - Alphanumerically ordered list of the pairs E, H, sorted according to H.
- G - The list of proposed updates u that producer p collects from other nodes in producer pool P. This is used to determine the most popular update.
- $\mathcal{L}(prod)$ - The list of producers from a particular producers' set G that sent the most popular ledger state update.
- L_{CE} - A producers list of compensation entries *i.e.* who a producer believes did the correct work for the network and thereby should be rewarded.
- $\mathcal{L}(vote)$ - The list of producers a specific producer found to have forwarded the correct vote for the most popular ledger state update.
- LSU - Ledger State Update. This is the final ledger state update proposed by a producer node. It consists of the list of entries used to create the update, the hash tree containing the signatures and the list of compensation entries.
- V - A producers vote for the correct LSU formed of the hash of the LSU, L_{vote} and the producer's ID.
- $\mathbf{o}$ - The final producer output that is declared to the network.

1.3 Construction Phase

The first phase of the Catalyst consensus algorithm is the Construction Phase. In this phase, the selected producer nodes P calculate either their proposed ledger state update.This is done by aggregating and validating all transactions that have occurred during a set time period. These transactions, assuming their validity, are integrated into the producers' local ledger state update, from which they can create a hash of the update. This hash digest represents what they believe to be the correct update and is broadcast to the other producer nodes for that cycle. Assuming the collision free nature of hash functions, the only mechanism for multiple producer nodes to

have the same local ledger state update is for both of them to use the same set of transactions.

Each producer p in the set of producers P follows the same protocol. The construction phase begins with producer p beginning their construction phase by flushing their mempool. This mempool T is made up of a set of transactions t, where the size of T is the number of transactions that have been broadcast to the network and have been stored by p. These transaction are used to create p's local ledger state update u.

The producer at this point also creates a hash tree d. This is to store the signatures that are extracted from each transaction in T. A salt σ is created utilizing a pseudo-random number generator using the previous ledger state update $U-1$ as its seed. p then follows the following steps:

1. Producer p verifies that each transaction in T is valid following the rules set out in [9]. From each of the n transactions in T the entries E that constitute the transaction t are extracted to form a list $E = e_1, ..., e_m$ for m entries in t. The producer should therefore end up with n lists for E from T. Each signature from the transactions is also extracted and added to d.
2. p for each E it created then creates a corresponding hash digest as:

   $$H = \mathcal{H}[E \mid\mid \sigma]$$

   Each pair (E, H) is added to a list L.
3. p then sorts list L into lexicographical order according to the hash values H.
4. The producer p then extracts the transaction fee value from each transaction in T to create v which is the total sum of all transaction fees.
5. The local ledger state update u for producer p can then be calculated. Firstly the list L is concatenated (denoted by $||$) with the hash tree d and a hash digest is created as:

   $$u = \mathcal{H}(L \mid\mid d)$$

   This list summary of transactions u is then concatenated with p's unique peer identifier Id to create:

   $$h = u \mid\mid Id$$
6. h is then broadcast to the other producer nodes on the network.

Producer p also collects other producers' partial ledger update values. At most they will collect $P-1$ values. Optimally every producer in P will receive the same set of transactions, so for every p in P they will have the same partial ledger update u. However this is unlikely due to all transactions not being received by a small group of nodes. Equally they may not hold G where $G = h_1, ..., h_P$, meaning they may not receive a proposed update from all candidates.

1.4 Campaigning Phase

The second phase of the consensus mechanism is where the producer p generates and proposes a candidate, which it calculates to be the most popular ledger state update.

Beginning this phase producer p has a set of partial ledger updates G that it has received from other producer nodes. Each h within G contains a producer's hash of the proposed update (u and a peer identifier (Id). The most popular u value can be found, which gives us u^{maj}, and from there the subset G_{maj} can be created, which is the amount of votes for the most popular update. Two thresholds must be considered first are G_{min}, the minimum amount of updates it has received from other producers in order to generate a valid candidate, and G_{thresh}, the threshold value for which a minimum number of votes must be in favor of G_{maj} (the most popular vote found within G). So in order to proceed with declaring a candidate the requirements, $G > G_{min}$ and $G_{maj} > G_{thresh}$ must be met.

If the thresholds are met the following can take place:

1. p creates a list $\mathcal{L}(prod)$. To this list p appends the identifier of any producer that correctly sent the u value that equals u^{maj}. If p's u value is also the same as u^{maj} then they should append their own Id.
2. Producer p then creates their candidate for the ledger update c which is calculated as $c = u^{maj} \mid\mid \#(\mathcal{L}(prod)) \mid\mid Id$
3. Producer p will then broadcast their preferred update c to the other producers.

p during this phase will be collecting the c values from other producers. At the end of this phase of the cycle p will hold a set of C candidates.

1.5 Voting Phase

The third phase of the ledger cycle is the Voting Phase within which a producer p from the C candidates it has received decides on what it believes should be the global ledger state update *i.e.* the update that should be applied to the ledger for that cycle.

1. p verifies that the same first hash value u^{maj} is embedded in a majority of producer candidates.
2. p is required at this point to have created the majority ledger update. As only with this partial update can they participate effectively. Therefore their $u == u^{maj}$

If each producer collects the first hash value generated by every producer, any two producers would build the same list of identifiers $\mathcal{L}(prod)$. However even in highly efficient distributed network it is unlikely that all producers will retrieve all information. Therefore it must be assumed that the list of identifiers held by p is incomplete, p however must ensure that they hold enough information to confidently issue a vote on the update to the ledger. The identifier of a producer is appended to the hash they distribute because:

- To verify that p is a producer node and that they in fact were one of the winners of the random draw described in **??**.
- To evaluate and track the quality of the work performed by p.
- To ensure that the correct producers withing P are rewarded for their work maintaining the ledger.

New transaction entries (compensation entries) are created using the list's $\mathcal{L}(prod)$ produced by the producers p. This allows reward and fees to be paid to the correct users. These new transaction entries will be appended to the final ledger state update for that cycle. Therefore a complete ledger state update should consist of:

- The list of transaction entries integrated into the new ledger state all E lists.
- The transaction signatures from T held in d.
- Compensation entries rewarding the producers from $\mathcal{L}(prod)$.

The voting process thereby must confirm the correct list of identifiers involved in producing the correct ledger state updates. The final list for $\mathcal{L}(prod)$ is generated by merging the producers lists together. In order for a producer p to have their Id added to the final $\mathcal{L}(prod)$ the must appear in at least 50% of the $\mathcal{L}(prod)$ lists distributed by P. This ensures that no producer can just append their own Id to their update whether that update was correct or not and gain a reward for it.

This process ensures that only producers that have provided the correct update may receive this part of the reward, thereby preventing unethical behavior from producers by declaring that they got the correct update when in fact they did not. However, it does introduce the problem of producers that have created the wrong ledger update being disincentiveise from proceeding in the consensus process further. Therefore producers that vote for the correct update whether or not they produced the correct update should also receive a portion of the reward. This is determined by the list $\mathcal{L}(vote)$ that each p creates to store the Id's of the producers which the gained votes from that voted for the correct update.

p follows a series of steps:

1. p generates a list $\mathcal{L}(vote)$ it then appends the Id of any other p that has forwarded a C candidate that satisfies $u == u^{maj}$.
2. p creates the combined list for $\mathcal{L}(prod)$ of any p from P that appears in $P/2$ lists.
3. Then p creates a list L_{CE}. This contains the contribution entries. This is for any user that is included in the combined $\mathcal{L}(prod)$. Each of these producers contained within this list will receive x tokens. If it is considered that X is the total number of tokens that are injected into the network at each cycle for the pool of P producers. The amount of entries into L_{CE} will be less than or equal to P. R is to be considered the total amount of reward that is to be split among all the producers that found the correct ledger update $R = f_{prod}X + v$ where f_{prod} is the fraction of the injected tokens that are given to users found in the combined $\mathcal{L}(prod)$. While $(1 - f_{prod})X$ is the value that is distributed to other nodes that have performed work.
4. Producer p can then create the candidate ledger state update for this cycle. This includes the reward allocations for producers according to their contributions. This ledger state update LSU can be defined as:

   $$\mathbf{LSU} = \mathbf{L} \mid\mid \mathbf{d} \mid\mid \mathbf{L_{CE}}$$

   p then computes its vote (or *producer vote*):

   $$\boxed{V = \mathcal{H}(LSU) \mid\mid \#(\mathcal{L}(vote)) \mid\mid Id}$$

5. p then forwards their V to the other producers and collects the producer votes V issued by its peers.

During the voting phase, the producer p collects the producer votes V broadcast by its peers. At the end of the voting phase , the producer p holds U producer votes V in its cache. This will be equal to or less than the number of producers who formed the correct update.

1.6 Synchronisation Phase

The Synchronisation phase allows those nodes that have produced the correct ledger update to then pass this update onto the rest of the network, thereby integrating the changes held within upon the overall state of the ledger.

During the synchronisation phase p executes the following steps:

1. Producer p selects from the producer votes V which received the most popular unique V. The number of most popular votes received is defined as V_{max}. This V_{max} value must exceed a specified threshold V_{thresh}. So $V_{max} \geq V_{thresh}$.
2. p creates a list $\mathcal{L}_{final}(vote)$. To $\mathcal{L}_{final}(vote)$ the producer p appends the identifier of any producer that their given $V == V_{max}$ *i.e.* they voted among the majority. Furthermore that producer must appear in at least half the $\mathcal{L}(vote)$ lists found in the V values p collected from other producers.
3. If the producer p also created the correct ledger state update LSU then they can write it to the DFS with the content address CID [**?**].
4. Producer p then creates the final producer output that can be declared to the network. This output is:

   $$\boxed{\mathbf{o} = CID \mid\mid \#(\mathcal{L}_{final}(vote)) \mid\mid Id}$$

   The producer then broadcasts $\mathbf{o}$ to the network.

2 Distributed File System

Running a full node on many blockchains requires a significant amount of disk space, for example the Ethereum blockchain size has exceeded 1Tb [10] and continues to grow. Catalyst solves this problem through the integration of a Distributed File System (DFS). This enables much more control for peers on the network what elements of the ledger that they hold. This means lightweight nodes can be ran. Furthermore, large data files can be securely held on the ledger without causing issues for other nodes due to bloating.

Catalyst integrates its own Distributed File System (DFS) [11] based on the InterPlanetary File System (IPFS) protocol [12]. DFS, as the name suggests, is a peer to peer file system, where the files contained within are distributed across and accessible to a range of peers on a network. This file system will have no down time nor any single point of failure (baring complete downtime of the internet), meaning that it is persistent in a way that other, centralised file storage technologies are not. IPFS is a peer to peer protocol that allows storage of files across many nodes, this is done by giving each item stored a unique identifier and providing a lookup mechanism pairing nodes that hold a item of data with the items unique identifier. Any duplicates are removed across the network, this is done as two identical files will always receive the same identifier. Using this identifier a file can be looked up on the network and retrieved from those nodes that are holding the file. Using this IPFS protocol as the basis for the DFS on Catalyst means that the information that is transacted on the network does not need to be stored 'on chain', meaning that there is a separation of DLT and stored data, allowing nodes to pick and choose what data they hold so any information not relevant to a particular node is not held.

On the Catalyst network every node is required to hold a DFS module. While all nodes will not hold all of the information stored on the DFS, they will be able to query and retrieve any information on the network. Furthermore, nodes will have the ability to query the existence of a file rather than being forced to download the file in order to check its existence. Nodes are required to hold the DFS module as it is used to store all information about the network. Some of the data secured on the DFS is the critical data that is used to sync the state of new nodes joining the network. This means that this data must be accessible at all times. Therefore, if a node wishes to check any single transaction on the network or even the overall state of the ledger itself it will have to be in sync with the network, which is done through the DFS.

Integrating DFS allows the Catalyst network to remain lean as nodes on the network can choose to store only the data that they choose to hold. Furthermore, it allows the storage of many rich file types in a distributed manner, meaning that they are always accessible with no down time guaranteed (excluding the potential case of the internet being down). This is further under the assumption that at least one node on the network holds the data. This also requires some nodes on the network to hold the file, if all nodes remove or unpin a file then that data will no longer be accessible to the network and must be re-added by a node. It is likely that if no nodes on the network require the file and therefore it is removed from the network that the file was redundent form use.

The Catalyst network allows peers on the network to

In this section is a description of the IPFS platform that Catalyst uses, and then how Catalyst integrates this into its ecosystem. Furthermore how the marketplace go the buying and selling of storage will operate.

2.1 IPFS

The InterPlanetary File System forms the basis of the Catalyst DFS. IPFS is a collection of protocols which specify how nodes are supposed to operate within the network, and various multilingual implementations of these protocols. As it is distributed, there is no one central entity that holds and controls the flow of information and most importantly who has access. Across the IPFS network nodes hold as little or as much data as the wish with multiple copies of the data

being held across many unrelated nodes. This means that there is no single point of failure in the network. An attempted attack on the IPFS would require a majority of nodes to be taken offline to prevent access to the files held on the network.

IPFS uses a bittorrent style seeding system. Meaning that the holder of any particular file that is online will seed the file to other peers that request the file. This means that the more peers on the network the more efficiently files can be distributed. IPFS user have the ability to 'pin' files meaning that those files are stored and secured by the peer. IPFS works in such a way that the more popular *i.e.* the more peers that have pinned the file, the more easily accessible that file will be.

In traditional computer file systems, files are indexed, referenced, and accessed through their location: for instance, `~/Documents/file.md` would refer to a Markdown file inside a "Documents" folder inside the home directory of a user on UNIX systems. However, on a distributed system, there can be no global state that could be used for addressing content. Instead of location-addressing, IPFS and other distributed systems use content-addressing. The CID also means that all files stored on IPFS are immutable i.e. can not be edited as changing anything contained within the file means that that CID for that file will now be changed. Through linkage of these files it means that a version history of files can be created.

In a content-addressed system, every file on the system is given a unique, deterministic content identifier (CID) by hashing the content. Hashing works by deterministically using the contents of the file as the input for a standard encryption algorithm. Changing the contents of a file would change the hash of the file. This CID can then be used to index, reference, and access the file directly, regardless of where it is. Advantageously, through content addressing, a malicious entity would not be able to send a user a false file in the place of the one that they have requested. This is due to the difficulty of reversing a hashing function. The CID associated with a file means that the file is not duplicated on the network. For example if Alice uploads file x it is stored on IPFS, other users can then download and hold the file, if then Bob uploads the same file x then because the CID for both files would be the same the duplication would not take place as the file is already held on the file system.

Once a file has a CID associated with it, a distributed hash table (DHT) can be formed. DHTs are key-value stores, where the key is the CID of the relevant file, and the value is an array containing all of the user IDs of any peer which holds the file. The DHT is used by peers looking for a file to find the peers that it can retrieve it from. Once a user has retrieved the file from another peer in the DHT, it can also then be added as a peer that holds the file.

IPFS integrates Merkle Directed Acyclic Graphs (Merkle DAGs). Merkle DAGs are abstract trees which result when multiple CIDs are used as the input for another hashing algorithm, resulting in chains where entire hierarchies of files can be referenced using a single CID. Any change to any constituent file at the bottom of the tree (the leaf nodes) percolates up to the next hash, and the next, causing the CID for the entire tree to change. Merkle DAG's are used compared to traditional Merkle trees as the allow a flow or direction of information to be established. Thereby allowing an order for the reading of the files to be set.

2.2 Integrating the DFS

Every node running on the Catalyst network must also be running the DFS module. This allows them on the most basic level to access the current state of the ledger and the validate previous updates to the ledger thereby allowing them to sync with the current ledger state. This is required as it is necessary for the all nodes on the network at the very least to know the current state of the ledger in order to send transactions.

The primary difference between the native IPFS protocol and the implementation utilised on Catalyst is how the peer IDs are created. While on the IPFS protocol the identifier for nodes is

selected randomly, on Catalyst the nodes on the network will each have their own individual peer identifier[13] made up of:

- IP address
- Port number
- Public key

This allows user to be able to identify who on the network is holding specific files as each peer identifier for a node will be unique. This means that a peer on the network will have a clear target for where they can retrieve given information, furthermore it means that nodes claiming to hold specific files and information are accountable for being able to distribute these files. Through the use of Catalyst peer identifiers within the DHTs peers can make informed decisions on who they retrieve files from, for example if a node has been marked as malicious due to actions within the network, they will also not be trusted within the DFS.

2.3 The Marketplace

WIP

The Catalyst network allows nodes to be rewarded for storing data added to the DFS by other nodes on the network. These may include large files or complex data sets. The reason for payment for holding of data is to provide incentive for peers on the network to store data for others. The market place works as a method of pairing nodes who wish to store data with those that have excess space they wish to sell. By incentivising the process of storage of data on the ledger it can be ensured that when uploaded a file is persistent and accessible at all times.

One key issue for this marketplace is ensuring that those nodes that are claiming to store a particular element of data are in fact storing it. Furthermore, it needs to be ensured that these nodes are online are respond to requests for the data in a timely manor. In some situations this data may need to be checked with a high frequency, for others it may be a long term storage solution. It must also be ensured that a particular storage node does not already hold the information that it is being requested to store, as this will lead to the node being paid twice for storing the same item.

Catalyst network utilises the Proof of Replication (PoR) and Proof of Space Time (PoST) mechanisms laid out by Filecoin [14]. Furthermore, through the use of the 0x protocol [15] so all storage purchase requests can be settled on chain.

PoR allows a storage node to prove to another peer on the network that a file given to it has been replicated and is unique. This means that this individual piece of data has been copied by the storage node and is held in a unique compartment of physical storage, *i.e.* that this is the first time this node has been asked to store this file and they are not getting paid twice for storing one file. This is important as the node storing the data on the DFS may (and in all likelihood will) want to store the data across multiple nodes to prevent a single point of failure. PoR prevents the storage node accepting multiple bids from the node storing their file on the DFS and taking multiple rewards for it.

PoST is an audit process for the storage nodes. It allows the creation of a challenge, the response to this challenge from a storage node allows it to demonstrate that it has stored the data required of it and furthermore that it is consistently storing the data without the need to repeatedly require it to present the data.

3 Global Ledger State

The global ledger state is the current state of all the accounts in the Catalyst Network. The state contains a key and value pair for every account. The 'key' is the account address and the 'value' contains account details such as the current account balance, and the number of previous transactions created from that account. A newly created account will only become part of the ledger state once it receives a transfer of Kats.

The global ledger state is not stored in the delta, instead a copy of the ledger state is stored by all participating nodes and updated each ledger cycle.

3.1 Account State

Due to Catalyst Network's extension of the Ethereum Virtual Machine into the KVM, the account information stored in the Catalyst account state is similar to an Ethereum account. Smart contracts and ordinary accounts use the same account structure, however some fields contain default values except in the case of a smart contract account.

The variables stored in the account state are:

Balance The number of Mol [16] owned by the account.

Commitment The private balance, represented by a Pedersen Commitment.

Nonce A number which is incremented when either public or private transactions are made from the account in order to prevent double spend attacks.

StorageRoot A hash representing the data stored by a smart contract.

CodeHash A hash representing the KVM code defining how the smart contract will operate.

In the Catalyst Network the the balance is stored directly in the state, rather than being derived from the transaction history as in Bitcoin. This has a number of advantages. New transactions can be validated more efficiently because it is faster to check that an account has sufficient funds. In addition, being able to store data state makes smart contracts much easier to design.

3.2 Ledger Storage

The global ledger state is stored locally by each node in a key value pair database. Any performant database such as RockDb can be used to store the state data, however the data must be organised using a structure called a sparse Merkle tree. This structure allows the entire data state to be represented by a single hash, known as the root hash. Any changes to the data would result in a completely different root hash, therefore it can be used by nodes to verify that their dataset is the same as that of other nodes. Consequently, a node using an alternative storage organisation would be unable to interact with the network due to an inability to calculate the same root hash.

The sparse Merkle tree data structure allows for:

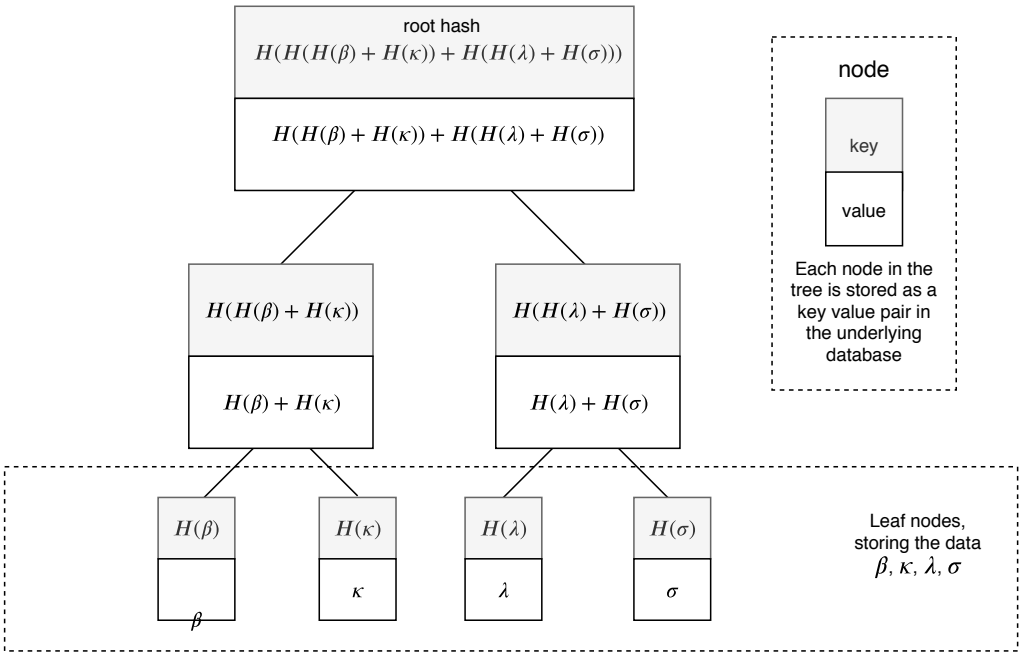
1. Persistent storage of the current and previous states of the ledger.
2. Efficient provision of proof that queried data exists, or importantly, does not exist in the ledger.

In addition to storing the global ledger state in a sparse Merkle tree, the smart contract KVM code and smart contract data are also stored using this data structure. This allows the root hash of both these data sets to be stored in the **storageRoot** and **codeHash** fields in the account state. The use of the SMT for storing these data sets allows honest nodes to recognise and reject information that has been altered outside of the ledger cycle by malicious parties.

3.2.1 Merkle Tree

Each item of data being stored is represented as a leaf of the Merkle tree. The data in neighbouring leaves is hashed and concatenated and this value is used for the node one level above in the tree. Each node is stored in the database with its lookup key being the hash of its value. The nodes are themselves combined until a single hash has been obtained. This is the root hash and its value depends on every bit of data stored in the tree.

Figure 1: A binary Merkle tree.



A binary tree is one in which the nodes can have only two children. Note that not all Merkle trees are binary trees. For example the nodes in a Modified Merkle Patricia tree can have up to 16 children.

When a data entry changes, every node above it in the tree must be recalculated. The hash of the data held by a node is used as its key in the database, therefore every recalculated node will constitute a new entry in the underlying database, rather than an existing entry that needs to be updated. This means that previous states of the tree are not overwritten, and all old data can be accessed using previous root hashes.

3.2.2 Merkle Proof

A client with knowledge of the current root hash of the state tree does not need to retrieve the complete set of data to verify the inclusion of a specific entry. Knowledge of each sibling hash along the path down to the entry being authenticated is sufficient to reconstruct the root hash. A client who is able to reconstruct the root hash from this data can be confident that the entry is contained in the data set. For example, in the diagram above, the inclusion of entry β can be verified with just β, $H(\kappa)$, and $H(\lambda) + H(\sigma)$.

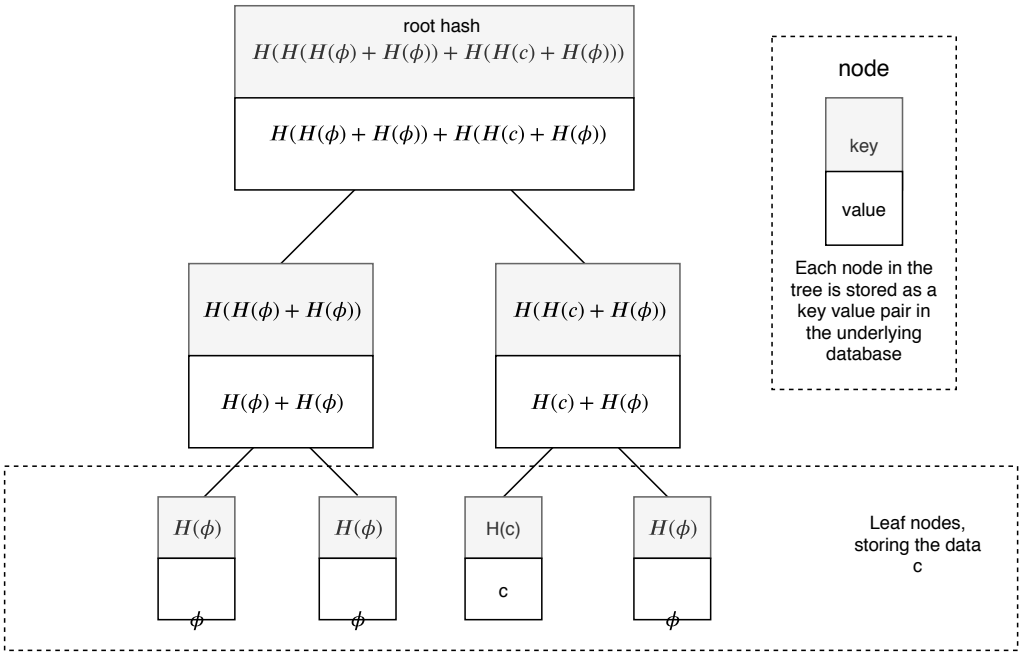
Though it is simple to prove inclusion of an item using a Merkle tree, proving that a particular item is *not* included can be harder. This is because the item could exist at any of the leaf nodes.

3.2.3 Sparse Merkle Tree

The Sparse Merkle tree is a binary tree. The position of an entry in the sparse merkle tree is determined by its hash. For the ledger state, we use the hash of the account address to determine the index of the leaf node to hold the account data. This account index is used to determine the path from the root hash to the data entry, telling us which branch to follow as we progress down the tree. The account index is not equivalent to a key in the underlying database.

In a sparse tree a default value is used for leaves that do not yet contain data.

Figure 2: A sparse Merkle tree, containing a single entry. The default values are represented by ϕ.



Since a data entry in a sparse Merkle tree can only exist in a specific location, proofs of non-inclusion are easy to provide; the proof that a key is not included in the tree is simply a proof that the value for the key is default.

A naive implementation would require a tree with a leaf for each possible index. Using the hash of the address as the index gives 2^{256} possible indices, too many to generate a full tree. However, non default entries will be uncommon compared to default entries; most of the nodes will therefore have default values that can be determined based on the height of the node. These values do not need to be stored more than once. For a branch containing only one non default entry, the entry can be stored in topmost node for that branch, providing further storage savings.

3.2.4 Choice of Sparse Merkle Tree over the Modified Merkle Patricia Tree

In a Patricia tree, each node can contain up to 16 children. As the sparse Merkle tree is a binary tree, it is simpler both to implement and to traverse the path to a leaf.

For the Patricia tree, the Merkle proof is composed of all the nodes in the path. Since each node has 16 children the amount of data is larger and not as straightforward to verify as in a binary tree. Using a sparse Merkle tree gives us Merkle proofs which are small and easy to use.

3.3 Updating the State

The current state of the ledger can always be obtained by applying the relevant deltas to a previous state. This process is deterministic such that applying the correct updates will always create an identical new state tree. Where data is unchanged the previous entries do not need to change in order to be included in the new state.

The merkle tree data structure has the additional advantage that as we apply each delta we do not overwrite information from previous ledger states. This means that the ledger history can be easily queried.

4 KVM

WIP

While smart contracts are capable of being ran on a wide variety of blockchains frequently they are simple and incapable of running complex solutions. Often also requiring a user to write smart contracts in a specialised language. Catalyst aims to solve this issue through the extension and improvement of the Ethereum Virtual Machine (EVM) to create the Kat Virtual Machine (KVM) [17]. By doing so, Catalyst will be compatible with existing smart contracts for the ethereum blockchain written in Solidity to be used on Catalyst while allowing more complex algorithms to be run far more simply than exists in the current infrastructure.

By extending the range of capabilities that are used on the EVM, we can ensure that significantly more complex operations can be performed by the Catalyst network. Furthermore, it will allow users of the network to have easily accessible cryptographic functions that are used on the Catalyst Network.

4.1 From the EVM

The EVM can be considered the Ethereum state machine. It provides nodes with an execution model that determines how the world state of the Ethereum network is to be altered given various byte-code commands. Nodes can thereby uniformly order actions that should be performed on the world state of the network. This allows any peer connected to the network to calculate what the current state of the network is.

It is structured as virtual stack, within which 1024 separate 256-bit values can be stored.

EVM is the run-time environment that enables smart contracts on the Ethereum blockchain. It relies on the fact that when a smart contract is ran upon two different machine using the same inputs both machines will retrieve the same output i.e. it is deterministic. Smart contracts can be written primarily in the Solidity language. This allows automatic settlement. This is possible as given a specific set of inputs the EVM will always return the same output, meaning that a smart contract ran by any peer on the network will always return the same result as all other peers.

It is considered Turing complete meaning that any algorithm that can be logically coded can be ran on the EVM. This is not strictly true due to Gas limit that restricts the number of operations that can be run by one contract, meaning that the there are large algorithms that require more operations that are possible with the current Gas limit.

Opcodes represent functions that can be performed by the EVM, and each opcode will affect the EVM's stack based infrastructure differently. Each of the opcodes has a price in Gas associated with. Gas is the currency used by the EVM for a user to pay for the cost of operations on the network, i.e. how much computing power is required to run that opcode. This is two fold, firstly it is used as a reward mechanism for miners on the network, secondly it prevents DDos attacks on the network from users spamming the network with difficult to run smart contracts with no financial repercussions. Examples of Opcodes include:

- STOP - Stops the contract running.
- ADD - Addition of the top two integers of the stack.
- MUL - Multiplication of the top two integers of the stack.
- SHA3 - Creates a Keccak-256 hash from a given value.

In total there are 140 unique opcodes, the full list of which can be found at [18]. Through these opcodes, any algorithm can be created and ran on the ethereum network providing it keeps to the Gas limit.

4.2 To the KVM

Smart contracts written for the EVM will be compatible with the KVM, meaning that no previously invested resources in developing Ethereum smart contracts will be wasted.

Precompiles are utilised in order to extend the functionality of the EVM. Precompiles or precompiled contracts are smart contracts that allow easy access to functions when writing smart contracts. These can be mapped to free slots in the namespace for the Opcodes. While these precompiles are not required to give Ethereum its pseudo-Turing completeness they enable users to produce smart contracts with a much higher level of business logic, meaning smart contracts are easier to create.

The KVM extends the range of Opcodes that are available on the EVM through the use of precompiles. These precompiles will allow interoperability with the Catalyst DFS, as some of the mathematical functions needed for confidential transactions. Precompiles are smart contracts that perform a function that are given input in

Examples of the additional pre-compiles that are added are:

- Range proof - Allows a range proof to be built
- Read DFS - Reads a part of the DFS according to its CID.
- Write DFS - Writes a file to the DFS, giving it a CID.

The use of precompiles rather than the addition of new opcodes means that KVM can retain interoperability with the EVM in the future while further extending its usability. Through the use of opcodes it also allows Catalyst to implement its specific cryptographic primatives that are used on the network as these will be different from those used on Ethereum. It also allows Catalyst to allow confidential transactions within smart contracts to be broadcast trivially. Interaction with the DFS integrated into Catalyst is also possible.

Conclusion

Catalyst employs a combination of existing technologies and novel techniques in order to create an infrastructure, that allows powerful storage and manipulation of data on the network, while also ensuring fair and easy accessibility for users.

Futher information on Catalyst and its applications can be found at:

Atlas City website - https://www.atlascity.io/
Catalyst website - https://www.catalystnet.org/
Catalyst GitHub - https://www.github.com/catalyst-network

References

[1] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] D. L. Chaum, *Computer Systems established, maintained and trusted by mutually suspicious groups*. Electronics Research Laboratory, University of California, 1979.

[3] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] A. Back *et al.*, "Hashcash-a denial of service counter-measure," 2002.

[5] P. Bernat, J. Kearney, and F. Sage-Ling, "Catalyst network research: a new consensus protocol." https://atlascitywebsitedocs.blob.core.windows.net/websitefiles/catalyst-consensus-paper.pdf, 07/11/2019.

[6] P. Bernat, J. Kearney, and F. Sage-Ling, "Catalyst network: Security considerations." https://atlascitywebsitedocs.blob.core.windows.net/websitefiles/Catalyst%20Network%20Security%20Considerations.pdf, 07/11/2019.

[7] B. Skvorc, "Two point oh: Randomness." https://our.status.im/two-point-oh-randomness/, 07/05/2019.

[8] randao, "randao." https://github.com/randao/randao/, 26/03/2019.

[9] catalyst network, "Transactionvalidator.cs." https://github.com/catalyst-network/Catalyst.Framework/blob/f23089d75ce876008ae1400f8044f538e07a3100/src/Catalyst.Core.Lib/Validators/TransactionValidator.cs, 07/11/2019.

[10] S. Hackernoon, "The ethereum-blockchain size has exceeded 1tb, and yes, it's an issue." https://hackernoon.com/the-ethereum-blockchain-size-has-exceeded-1tb-and-yes-its-an-issue-2b650b5f4f62, 19/05/2018.

[11] catalyst network, "Catalyst.core.modules.dfs." https://github.com/catalyst-network/Catalyst.Framework/tree/develop/src/Catalyst.Core.Modules.Dfs, 07/11/2019.

[12] J. Benet, "Ipfs - content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[13] catalyst network, "Bytesextensions.cs." https://github.com/catalyst-network/Catalyst.Framework/blob/b761153e5e30a65b843d470cf60175256bd43276/src/Catalyst.Core.Lib/Extensions/BytesExtensions.cs, 07/11/2019.

[14] J. Benet and N. Greco, "Filecoin: A decentralized storage network," *Protoc. Labs*, 2018.

[15] W. Warren and A. Bandeali, "0x: An open protocol for decentralized exchange on the ethereum blockchain," *URl: https://github. com/0xProject/whitepaper*, 2017.

[16] catalyst network, "Catalyst.core.modules.kvm - catalystunit.cs." https://github.com/catalyst-network/Catalyst/blob/master/src/Catalyst.Core.Modules.Kvm/CatalystUnit.cs, 07/11/2019.

[17] catalyst network, "Catalyst.core.modules.kvm." https://github.com/catalyst-network/Catalyst.Framework/tree/develop/src/Catalyst.Core.Modules.Kvm, 07/11/2019.

[18] ethervm, "Ethereum virtual machine opcodes." https://ethervm.io/, 28/09/2019.