Atomchain Storage: A distributed-decentralized storage (ddStorage)

Author: Okpara Okechukwu D.
Email: okpara.net@gmail.com
Website: www.okpara.net
Date: 01-05-2022

Abstract:
We propose a method of distributed decentralization of files, data structures, and persistent algorithms for atomchain technology, which aims to eliminate the inefficiencies of a traditional decentralized storage (dStorage).

Keywords: atom network, atomchain, dStorage, ddStorage

1. Introduction

Decentralized storage systems consist of a peer-to-peer network of user-operators (or nodes) that hold a portion of the overall data, creating a resilient file storage sharing system. All data of atomchain is stored mainly in active nodes. These nodes are connected through the P2P protocol.

When a user initiates a transaction, the transaction is broadcasted through the P2P protocol, and the atom creator (or miner) node will verify it, package it, and then broadcast it to the atom network.

If atomchain were to grow steadily, expanding to large amounts of data it will not be feasible for all atomic (reactor) nodes to continue to run. And the cost of deploying such huge data in the future to Atomchain's Atom Network would be prohibitively expensive due to computation energy fees.

2. Distributed-Decentralized Storage (or ddStorage)

This involves the combined effort of applications on atom networks running atomchain protocols. Atom network is both decentralized (there is no central node or server that controls the operations of the network) and distributed (multiple nodes work together, for instance by sending messages).

Atomchain’s ddStorage service is like a decentralized program running on a distributed platform using trie data structures to manage data [1]. We shall delve into ddStorage mechanisms by considering four aspects of a decentralized storage (dStorage):

- Persistence mechanism / incentive structure
- Data retention enforcement
- Decentrality
- Consensus

3. Persistence Mechanism / Incentive Structure

Persistence mechanism is needed for a piece of data to persist permanently. Here, the whole chain needs to be accounted for when running an atomic (reactor) node. A new piece of data is bonded

into the last atom (atomsphere), and it continues to grow – requiring every node to replicate all the encapsulated or embedded data. This persistence Mechanism has an incentive structure, in which a cryptocurrency payment is made to the atomsphere creator. The atom creator (or miner) node is paid to add the data on the atomchain. Moreover, quark objects (using omega atom wallet addresses) can be used to store the hash of the location of a piece of data. To keep the data persisted, these quark objects or codes must be continually funded.

4. Data Retention Enforcement

To ensure data retention, Atomchain is expected to use some type of cryptographic challenge that is issued to the nodes to make sure they still have the data. Here Atomchain issues a challenge to the active nodes to see if they have the data at both the most recent atom and a random atom in the past. If a node is found wanting or deficient, it is penalized.

5. Decentrality

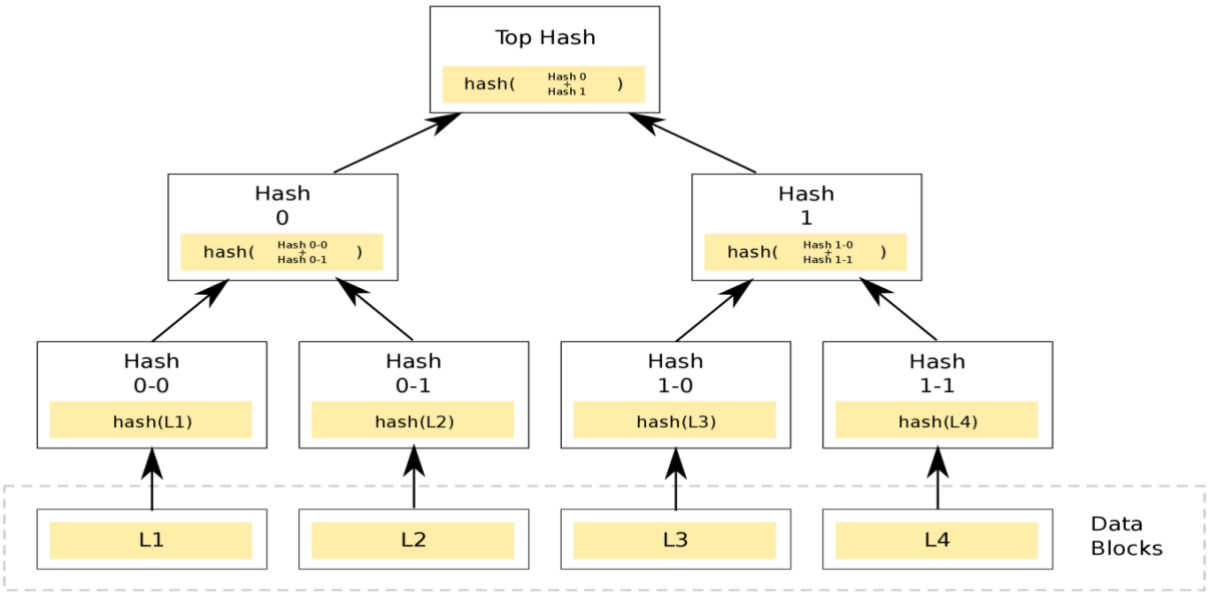
Atomchain decentralization may be measured by an application that can account for the entire atom network. In particular, a decentralized application that can check a given cluster of atom fields' analytics and metrics.

6. Consensus

Atomchain main consensus mechanism is based on atomic bonding.

7. Transaction Storage

According to Wikipedia [2], a Merkle tree is typically a binary tree typical binary tree.



Hash trees allow efficient and secure verification of the contents of large data structures. Hash trees are a generalization of hash lists and hash chains. Therefore, using a Merkle tree for our transactions will allow atomchain to be as compact as possible in terms of disk space, as well as allowing for secure verification of transactions.

From the Atomchain Network Whitepaper [3], atomic nodes will contain the whole transaction history while orphaned atomic nodes will contain the Merkle root hash.

8. Ω Address Transaction Events

In atomchain, there two types of transactions:

1. Normal α transactions, and
2. Ω transaction events

A transaction event is a cryptographically signed instruction from an Ω address, which changes the state of the atomchain.

These transactions are "**events**" that can trigger a change of state or cause the Atomchain Virtual Machine to modify the state of the atomchain, for instance, by evaluating orbit codes or updating balances of the Ω addresses.

A transaction event is usually a byte-array signed message originated by a Ω address at rest, transmitted by the atom network, and recorded on the atomchain.

α-based	Ω-based	FIELD	BYTES	DESCRIPTION
		ver	4	Version
as TXID		Transaction index (TX_{id})	4	
as TXcount	as ΩEVT_{cnt}	Event count (TX_{cnt})	16	Number of transaction events from the sender. It starts from 0 and increments each time a transaction is sent or each time a transaction event occurs
		energyPrice (E_{price})	16	
		energyMax (E_{max})	16	The maximum number of energy units that the transaction event

				will be allowed to consume
as (TX_{out})	as 20-byte Ω address	Recipient address (TX_{out})		This is the address to which the transaction is being sent. For Ω addresses at rest, the transaction will involve a transfer of value. For Ω addresses in orbit, the transaction will result in the orbit's code being executed
		O_c or O_d	varies	Message call data or orbit code creation data
		T_v (uint8)	1	v of ECDSA signature output (x-coordinate) Note: signature is data that identifies and authenticates the transaction's sender
		T_r char[32]	32	r of ECDSA signature output (recovery)
		T_s char[32]	32	s of ECDSA signature output Note: that ECDSA elliptic curve is secpk256k1
		data		For orbit code related activities such as deployment or execution of a code. During code deployment, this optional arbitrary binary data is where the orbit byte code is sent. When calling an orbit code, this

				specifies which functions should be called and with what arguments
as little-endian integer		Amount/Value (TX_{value})	32	The amount of oke to transfer from the sender to the recipient address. This amount may be zero.
		maxFeePerEnergy		The max energy fee a user is willing to pay for the transaction to be processed
		maxCreatorEnergyFee		The amount of energy intended to serve as a tip to the miner who processes the transaction
`TXin`		`TXin`	varies	
`TXout`		`TXout`	varies	
`inCount`		`inCount`	varies	
`outCount`		`outCount`	varies	
		timeLock	2	a reference to timestamp in unix epoch or atom index
as Shell		coinbaseShell	50	`Arbitrary data. Miners commonly place TXcount in this field to update the nucleus merkle root during hashing`

Table 1. Data Structure of atomchain transactions (and events)

The atom network serialization format is the only standard form of a transaction event, hence, a transaction event can be seen as a serialized binary message or byte array that contains the following data: event count, energyPrice, energyMax, recipient address, value, data field and (v,r,s).

event count is a sequence number of transaction sent from the address. Each time a transaction event is sent from the address, the event count value is incremented by one. Event count helps in preventing message replay attacks.

energyPrice represents the price of energy in atomcoin the originator is willing to pay, for instance, 1 energy = 10 atomcoins. It is determined by market supply and demand.

The price of energy = energyPrice * energyMax

energyMax is the limit of the amount of OKE the sender is willing to buy for this transaction event. If energyMax is not enough to transfer OKE, the transfer will be cancelled and energy will be refunded to the sender. If the energyMax is set in excess, the leftover energy will be refunded to the sender.

Recipient address is the destination Ω address either at rest or in orbit.

Value represents the amount of OKE/atomcoin from the originator to the recipient. Value is used for both transfer of money and orbit code execution.

Data field is used for deployment or execution of orbit codes. It is a variable-length data payload. Messages in data field can be seen as function calls. If this field is empty, it means a transaction event is for a payment not an execution of orbit code.

A transaction event needs to contain messages in order to call/execute functions. A message can contain specific details about the action being authorized and any parameters required to execute the action.

A **message** contains: the sender of the message, the recipient of the message, the value field which contains the amount of atomcoin to transfer alongside the message to the orbit code address, optional data field that is the actual input data to the orbit code, **initEnergy** value that limits the maximum amount of energy the code execution triggered by the message can incur.

v, r, s are the components of an ECDSA digital signature of the originating Ω address at rest. ECDSA is used as a digital signature for verification. v indicates both the atomID and the recovery ID to help the **ECDSArecover** function check the signature. r and s are inputs of ECDSA and **sec256k1** constants to define the elliptic curve. The signature identifies the sender. The signature is generated when the transaction is signed by the sender's home.

The state of the atomchain can be changed when a Ω address transaction event is executed on it. This event is like what happens between the states in atomchain, such as keeping track of the address balances and details of different addresses at any point in future time.

Ω address transaction events (or Ω transactions, in short) are stored in form of **events root**, **evanescent root** and **things root**. The trie's root node (events trie, evanescent trie or things trie) is a unique, deterministic cryptographic hash that can be used as evidence that the trie has not been tampered with.

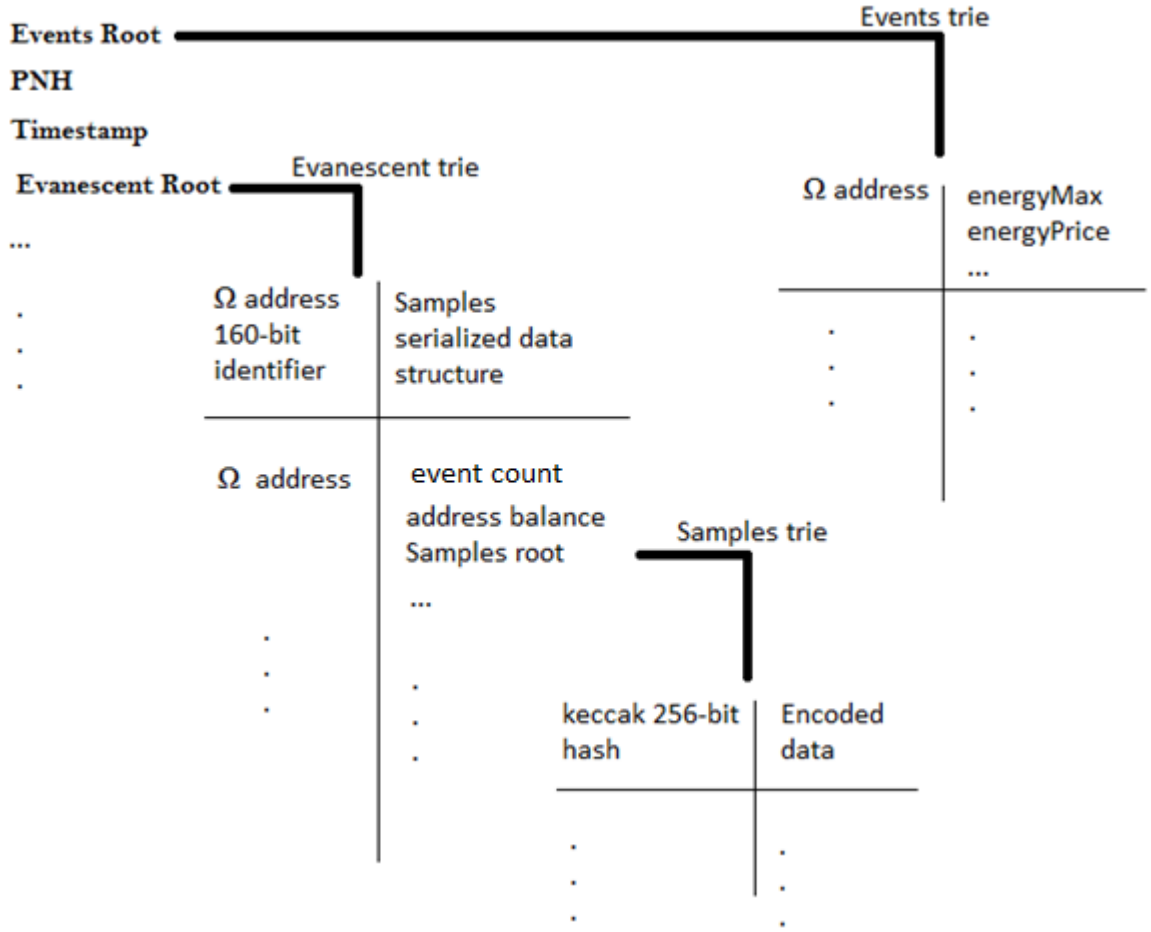
Events trie is a permanent data type; example is an event that has been fully confirmed. The keccak-256-bit of the events trie's root node is referred to as the **events root**.

Evanescent trie is a global transitory data type that is constantly updated; example is the balance of an Ω address. Evanescent trie contains all **Ω addresses and samples pairs** that exist on the atom network. Ω addresses are 160-bit identifiers. Samples are created by encoding the event count, address balance, virtual root, and codeHash.

Evanescent trie's root node (referred to as the **evanescent root**) is a hash of the entire evanescent trie at any given point in time and it used as a secure and unique identifier for the trie. An evanescent root node is cryptographically dependent on all internal evanescent trie data.

In a given atom, the **keccak-256-bit hash** of the evanescent trie's root nodes is stored as the evanescent root value.

Each Ω address has its own samples trie. A 256-bit hash of the samples trie's root node (referred to as the **samples root**) is stored as the virtual root value in the global evanescent trie.



Ω address are only added to the events trie only when a successful transaction event has taken place with respect to the Ω address in such that energy was consumed when the event was included in the atom; this guard against malicious attackers trying to continuously bloat the events trie.

An events trie of an atom in an atomchain contains many transaction events. The order of these events is decided by the atom creator that assembled or mined the atom. mined atoms are never updated, and the position of the transaction events within the atom is permanent.

Finally, in order to reference a particular trie in an atom, one needs to obtain the trie's root hash as a reference.

9. α Address Transactions

If you want to verify α transaction ID (TXID) or the raw transaction data of a given transaction, you calculate the double-SHA256 hash of the raw transaction data which should match the transaction ID:

```
TXID == SHA256(SHA256(raw_transaction))
```

A coinbase α transaction is a special transaction created by the creator of the atom. It is used for the atom creator to collect the mining reward and transaction fees from other transactions in the same atom. Each atom should have only one coinbase α transaction. And it is placed as the first transaction in the transaction list.

10. Transaction (Event) Signing

1. Calculation of the unsigned transaction's hash (uTX_{hash}): this calculates the Keccak-256 hash of the byte array from ver to TX_{value} (inclusive) from Table 1.
   uTX_{hash} = KEC256(ver, … , TX_{value})

2. Signing of the calculated hash: this is the deterministic signing of the using a home to output the values of v, r, s.
   ECDSA(uTX_{hash},home) = (v, r, s)

After a transaction is signed and submitted, a series of events takes place:

1. A transaction hash gets cryptographically generated.

2. The transaction is broadcasted out to the atom network in a pool of numerous other transactions.

3. A miner selects the transaction and includes it in the next atom to verify the transaction.

4. The transaction receives “confirmations.” Each confirmation equals one new atom of the particular (α- or Ω-) operator created since the atom that the transaction was a part of. The more confirmations, the more certain it is that the transaction will be “successfully” processed by the network.

Generally, a transaction (event) can be considered final after 4 atoms.

11. Conclusion

In the future, atomchain ddStorage may also support compatible content distribution and file referencing dStorage platforms for storing data; as well as multi-chain ddStorage compatibility.

12. Definitions

- A **binary tree** is a tree whose elements have at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

- A **Merkle Tree** is a tree in which every leaf node is labelled with the cryptographic hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes. For example, in the picture hash 0 is the result of hashing the concatenation of hash 0–0 and hash 0–1. That is, hash 0 = hash(hash(0–0) + hash(0–1)) where + denotes concatenation.

- A **tree** is a collection of nodes, where each node is a data structure consisting of a value, together with a list of references to nodes.

13. References

[1] Okpara O. D. 2022. The Atomchain Wallet Whitepaper, https://okpara.net/AtomWallet.pdf

[2] Retrieved on 01-05-2022 from https://en.wikipedia.org/wiki/Merkle_tree

[3] Okpara O. D. 2022. The Atomchain Network Whitepaper, https://okpara.net/AtomNetwork.pdf