**mkx**

**A Blockchain Data Storage Service Network**

Javed Dilshad Neemuth (dil.neemuth@gmail.com)

**Abstract**

A decentralized storage framework implementing client-side encryption that would allow users for time stamping and fingerprinting any data objects, files and streams.

The removal of central controls would mitigate most traditional data failures and outages, as well as significantly increase security, privacy, and data control. Peer-to-peer networks are generally unfeasible for production storage systems, as data availability is a function of popularity, rather than utility. We propose a solution in the form of verification and reward system to encourage data storing. In this way we can periodically check data integrity, and offer rewards to peers maintaining data. We further propose a model for addressing access and performance concerns with a set of independent or federated nodes.

The proposed framework would be built in such a way as to facilitate integration with existing legacy systems and achieve an acceptable level of trust when sensitive data is being shared. This whitepaper discusses the problems that existing data transfers involve and how the framework resolves them.

**Keywords:** Data Protection, Encryption, Data Ownership

**1. Introduction**

Under the original Bitcoin whitepaper, the transactions effected through the network were used to act as a store for value to depict cryptocurrency. This principle was replicated in all subsequent developments which resulted in a decentralized peer-to-peer online currency that maintains a value without any backing, intrinsic value or central issuer.

The traditional storage system is vulnerable to a variety of security threats, including man-in-the-middle attacks, malware, and application flaws that expose private consumer and corporate data. Moreover, as all data resides on a central server on same infrastructure; this results in a single point of failure.

A decentralized exchange service offers many advantages compared to datacenter-based cloud storage. Data security can be maintained using client-side encryption, while data integrity will be maintained via a proof of availability. The impact of infrastructure failures and security breaches will be greatly reduced. An open market for data storage may drive down costs for various storage services by enabling more parties to compete using existing devices.

Data on the network will be resistant to censorship, tampering, unauthorized access, and data failures. This paper describes a concrete implementation of such a network, and a set of tools for interacting with that network.

**2. Design**

mkx is a blockchain solution that creates a distributed network for the data storage between peers. The mkx protocol enables peers on the network to transfer data, verify the integrity and availability of remote data retrieve data, and extract data from transactions. Each peer is an autonomous agent, capable of performing these actions without significant human interaction. Many of the basic tools for these interactions are described in this document.

**2.1 Digital Data as encrypted fragments**

A fragment is a portion of an encrypted file/object to be stored on this network. Fragmentation has a number of advantages to security, privacy, performance, and availability.

Files should be encrypted client-side before being fragmented. The reference implementation uses both symmetric (AES) and asymmetric (RSA), but convergent encryption or any other desirable system could be implemented. This protects the content of the data from the storage provider, or other peers, housing the data. The data owner retains complete control over the encryption key, and thus over access to the data.

The data owner may separately secure knowledge of how a file is fragmented and where in the network the fragments are located. As the set of fragments in the network grows, it becomes exponentially more difficult to locate any given fragment set without prior knowledge of their locations. This implies that security of the file is proportional to the square of the size of the network. Fragment size is a negotiable contract parameter. To preserve privacy, it is recommended that fragment sizes be standardized as a byte multiple, such as 8 or 32 MB. Smaller files may be filled with zeroes or random data. Standardized sizes dissuade side-channel attempts to determine the content of a given shard, and can mask the flow of shards through the network.

Because peers generally rely on separate hardware and infrastructure, data failure is not correlated. This implies that creating redundant mirrors of fragments, or applying a parity scheme across the set of shards is an extremely effective method of securing availability. Availability is proportional to the number of nodes storing the data.

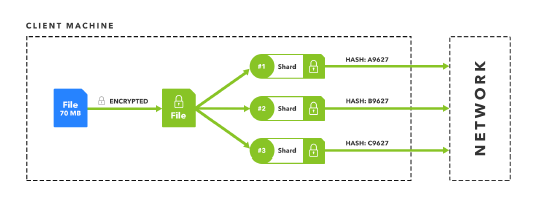
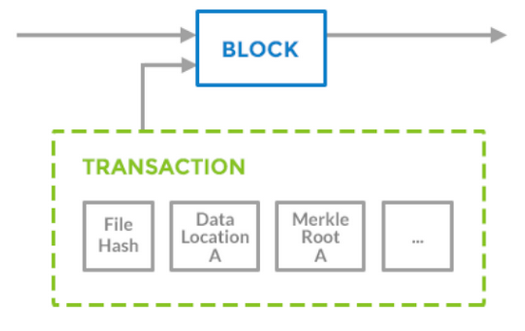


Figure 1: Visualizing the Fragmentation Process

1. Files are compressed and encrypted.
2. Encrypted files are split into fragments, or multiple files are combined to form a fragment.
3. Pre-processing is performed for each fragment ( see Section 2.3 )
4. Fragments may be transmitted to the network.

**2.2 Swarming**

The content data is not stored on the blockchain itself; as this is very expensive in terms of network bandwidth. Instead, the content is compressed using GZIP format; divided into a set of fragments which contain partial data. Only the swarm data of these nodes are stored in an encrypted format in each transaction on the blockchain.



1. Transaction is published to network with content data.
2. The content data is compressed in GZIP format; split into multiple fragments which are then sent randomly to any active nodes ( this is done randomly to minimize access to known nodes )
3. The nodes receiving the fragments; the partial content hash of each fragment and index are tracked.
4. In order to relocate the scattered information over network; the following fields are added to the swarm related info: node url, partial hashed content & index.
   1. The swarm upload information is denoted as follows : node url **SPACE** partial hashed content **SPACE** fragment index

e.g. <http://localhost:8080/node/> 734asdd456rfrg 0

5. The fragments are stored on each node instance in a local store database such as MapDB.

**2.2 REST Architecture**

The networking part of mkx is built on REST architecture, an efficient communication protocol for nodes to communicate with each other. The nodes communicate with a master node to build and scale an efficient p2p network.

**2.2.1 Signature Verification**

Similar to Bitcoin, the mokachain network requires peers to sign messages. To join the network a node must create an ECDSA keypair, (Kpriv; Kpub). The mkx Node ID corresponds to the hashed name and public key e.g. SHA256 (Kpub+name)). As such, each Node ID in the mokachain network is also a valid public address, which the node can spend from. Nodes sign all messages, and validate message signatures before processing messages. This modification enforces long-term identity on the network, and provides a proof of work deterrent to Eclipse attacks. In the future there are a variety of other uses for this address.

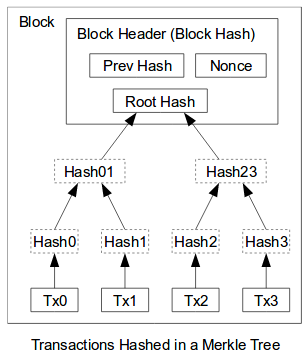
**2.3 Proofs of Availability**

Proofs of availability guarantee the existence of a certain piece of data on a remote host. The ideal proof minimizes message size, can be calculated quickly, requires minimal pre-processing, and provides a high degree of confidence that the file is available and intact. To provide knowledge of data integrity and availability to the data owner, mokachain provides a standard format for issuing and verifying proofs of availability via a request-response interaction called **node heartbeat algorithm**.

Our reference implementation uses Merkle trees and Merkle proofs. After the fragmentation process, the data owner generates a set of n fragmented sets of f0; f1; ... fn-1 and the swarm merkle root is calculated for the number of fragments generated.

A request is sent for each transaction at a scheduled time to check whether the fragment hashes match with the related swarm root hash. This process ensures the following functions:

1. Each node contacted is active and serving the fragmented data.
2. The integrity of the whole data is intact and has not been tampered with.



The swarm merkle proof always consists of exactly log2(number of leaves) + 1 hashes, and thus is a compact transmission, even for large trees. The data owner uses the stored Swarm Merkle Root and tree depth to verify the proof by verifying that its length is equal to the tree depth and the hashes provided recreate the stored root. This scheme does not allow false negatives or false positives, as the hash function requires each bit to remain intact to produce the same output.

* + 1. **Node heartbeat algorithm**

The node heartbeat algorithm is a mechanism to ensure that the nodes hosting the partial fragments are active and ensuring the distribution of the stored data to other nodes. User requests transaction information on the mkx network through a request transaction id.

1. At each interval; a request is sent to n number of nodes to check whether they are still active.
2. If the node is active; it sends back a response with an updated timestamp.
3. The list of active nodes is updated with recent timestamps else the inactive nodes are removed from network until they re-log on the network.
   * 1. **Reward System**

In the mkx protocol; we need to encourage users to setup mining nodes to process and validate transactions. The more nodes; the better as security will be re-enforced by the network effect of these nodes. Below is a reward proposal system that has been planned.

The reward system in mokachain works in the following order.

Hosting nodes:

1. A client sends a request for transaction on the mokachain network.
2. The nodes communicate between themselves to find the corresponding set of data fragments through the swarming technique.
3. Each time a node serves a data fragment; the concerned node is rewarded with a certain amount of coins.

The reward sum if defined by two factors:

1. The uptime the node has been on the network.
2. The load of data that the node is serving.

Mining nodes:

The node is also rewarded when it succeeds to mine a whole block.

**2.4 Simple Mirroring**

The simplest solution is to mirror shards across several nodes. Mirroring protects against hardware failures by ensuring that multiple copies of each shard exist. Availability of the shard with this scheme is



where an is the uptime of the node storing fragment n. Because all shards are required to assemble the file,

availability of the file is equal to the availability of the least available shard. In the case of a dropped contract, a redundant copy of that shard can be retrieved and a new location found for it on the network. This is the current behavior of the reference implementation.

In the current version; if one of the nodes is not available; the swarming service will try to contact the federated nodes

to obtain the missing shards in order to reconstruct the whole file as is.

As a rule of thumb; the total numbers of shards are kept exceptionally on the federated nodes for recovery use cases.

**2.5 K-of-M Erasure Coding**

mkx will soon implement client-side Reed-Solomon erasure coding. Erasure coding algorithms break a file into k shards, and programmatically create m parity shards, giving a total of k + m = n shards. Any k of these n shards can be used to rebuild the file or any missing shards. Availability of the file is then across the set of the m + 1 least available nodes



. In the case of loss of individual shards, the file can be retrieved, the missing shard rebuilt, and then a new contract negotiated for the missing shard. To prevent loss of the file, data owners should set shard loss tolerance levels.

Consider a 20-of-40 erasure coding scheme. A data owner might tolerate the loss of 5 shards out of 40, knowing that the chance of 16 more becoming inaccessible in the near future is low. However, at some point the probabilistic availability will fall below safety thresholds. At that point the data owner must initiate a retrieve and rebuild process.

Because node uptimes are known via the audit process, tolerance levels maybe optimized based on the characteristics of the nodes involved. Many strategies may be implemented to handle this process.

Erasure coding is desirable because it drastically decreases the probability of losing access to a file. It also decreases the on-disk overhead required to achieve a given level of availability for a file. Rather than being limited by the least available shard, erasure coding schemes are limited by the least-available n + 1 nodes.

**2.5 MapDB persistence storage**

To facilitate on-disk storage for nodes, mkx implements a key-value store called MapDB.

MapDB provides Java Maps,Sets, Lists, Queues and other collections backed by off-heap or on-disk storage. It is hybrid between java collection framework and embedded database engine.

All the participating nodes have an inner mapdb database; where the data fragments are stored. mapdb is an abstraction layer over a set of traditional hashmap instances that seeks to address scaling problems.

**2.5.1 Rationale**

MapDB is a key-value store. It has many desirable qualities, including long-term support, portability, and high performing reads powered by lexicographically sorted keys. While MapDB compaction is typically a desirable feature, it severely limits scaling. Its impact is larger on lower end systems and can also vary based on the type of disk in use. Compaction also blocks reads and writes during this period, rendering mokachain nodes effectively offline until the process completes. In order to circumvent this limitation; the content byte is already compressed in GZIP format for storage.

However, because MapDB instances are cheap to create, open, and close, compaction costs can be bounded by managing a set of size-limited MapDB instances. Instances can be initialized ad hoc, and opened and closed as necessary.

**2.5.2 Keying By Hashed Composite Key**

As mentioned earlier, MapDB sorts items lexicographically by key. MapDB takes advantage of this to optimize the efficiency of reads and writes. Data is stored in chunks of C bytes (or less). By default C = 128 KiB. These chunks are keyed by the full content's hash followed by a space and a numerical index. This ensures that key/value pairs are small, and that reads and writes to and from a transaction is sequential. It also allows for efficient streaming of data both in and out of the transaction.

Because of the idiosyncrasies of sorting numerical strings lexicographically, the index substring should be expressed in base 10, and have a constant number of characters. The length l of the index string should be defined by



For default parameters l = 6. Therefore the chunk at i = 3753 will have the index number '003753'. This ensures that chunks of a shard are stored consecutively.

To preserve order and therefore maximize read/write performance, the keys for the chunks of a specific shard should be strings generated as follows:

1. Append the swarm node url where fragment is being sent to the string
2. Add single space
3. Append the partial hashed data for the fragment to the string
4. Add single space
5. Determine the chunk index number, and append to string.

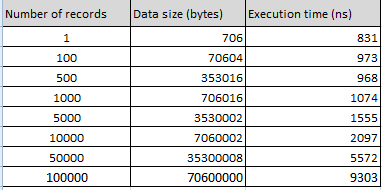
6. Encode SHA256-Hex (data) as a string of hexadecimal characters.

7. Append the mod

**2.5.3 Performance Benefits**

Discuss on performance of mapdb (add more graphs… additional tests to be done)

A series of tests have been carried out with the local persistence store. Please note that a text file was being used to keep information and the record lines were gradually being added.



**2.5.4 Data Transfer**

Data is transferred via HTTP. Each node exposes endpoints where client applications may upload or download fragments. Clients' requests are authenticated via tokens provide by generated key api. This transfer mechanism is not essential to the protocol, and many alternatives may be implemented in the future.

**2.5.4.1 Transacting with minimum data**

One of the key aspects of the framework is DATA MINIMISATION, whereby the amount of information that is shared by a user to counterparty is restricted to a strict minimum to reduce the risk of unauthorized access and other impersonation threats. Built on the validation process; this function makes use of the results derived from the consensus.

**4 FUTURE Areas of Research**

mkx is a work in progress, and many features are planned for future versions. There are relatively few examples of functional distributed systems at scale, and many areas of research are still open.

Considering other alternatives for local persistence which would comprise a high-level of encryption .e.g. work on **MFS (Mkx File System)** already started.

One particular area to consider is to move away from the **proof of work** concept as it is too hardware-resource intensive.

**3. Transaction Analysis**

**Do some stats ( with charts )**

1. **Conclusion**

We have proposed a system for data exchange and storage without relying on trust. We started with the usual framework containing sensitive digital content made from digital signatures, which provides strong control of ownership, but is incomplete because the data cannot be stored in the blockchain. To solve this, we proposed a peer-to-peer network using proof-of-work to record a public history of transactions that quickly becomes computationally impractical for an attacker to change if honest nodes control a majority of CPU power. The digital content is fragmented and scattered over a number of nodes thus limiting attackers from obtaining the whole dataset. Only clients with suitable private key are able to retrieve and assemble the original piece of digital data from the network. Any needed rules and incentives can be enforced with this consensus mechanism.

References

[1] Nakamoto, Satoshi. “Bitcoin: A peer-to-peer electronic cash system.” (2008)

[2] Lamport, Leslie, Robert Shostak, and Marshall Pease. “The Byzantine generals problem.” ACM

Transactions on Programming Languages and Systems

(TOPLAS) 4.3 (1982): 382-401.

[3] Attiya, C., D. Dolev, and J. Gill. “Asynchronous Byzantine Agreement.” Proc. 3rd. Annual ACM

Symposium on Principles of Distributed Computing. (1984)

[4] G. Hall. Storj core tutorial: Protocol specification, (2016).

<https://storj:github:io/core/tutorial-protocol-spec:html>.

[5] H. Shacham, B. Waters. Compact proofs of retrievability, (2008).

<https://cseweb:ucsd:edu/~hovav/dist/verstore:pdf>.

[6] R.C. Merkle. Protocols for public key cryptosystems, (April 1980).

<http://www:merkle:com/papers/Protocols:pdf>.

[7] J. S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems, (1996).

http://web:eecs:utk:edu/~plank/plank/papers/CS-96-332:pdf.