2019 International Conference on Advancements in Computing (ICAC)
December 5-6, 2019. Malabe, Sri Lanka

VAULT - A Shared Distributed And Redundant Storage Solution

T.R.N.R. Peiris, W.M.U.K.M.T. Bandara, K.V.A. Sachintha, AmilaSenarathne, B.A. Ganegoda
Faculty of Computing, Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
peiris1996@gmail.com, ukmihiran@gmail.com, amod077@gmail.com, amila.n@sliit.lk, asviganegoda@gmail.com

***Abstract*—An ideal distributed storage solution must have the ability to provide redundant, reliable, shared and secure access to user data without compromising the ability to scale and descend while maintaining performance. VAULT is an attempt to avert the negatives of the cloud in a local environment using a decentralized methodology. VAULT makes use of individual idle storage space on a network of peer-to-peer nodes which is then provided to an end user to store files in the pooled space. VAULT implements redundancy by the use of Reed-Solomon codes and maps file fragment locations using a blockchain as a distributed ledger. Fragment distribution is optimized using a machine learning approach where node characteristics are used to determine the reliability of each node. The aggregation of above features makes VAULT and ideal solution for corporate environments where consumer hardware and infrastructure is already allocated.**

Keywords—Reed-Solomon, distributed-ledger, peer-to-peer, distributed-storage, Merkele-trees

I. INTRODUCTION

A huge amount of storage space sits idly unused in mobile devices, hard drives etc. in the world. Large scale corporations that routinely use thousands of workstations for their employees have hundreds of terabytes of unused free space in their hard drives. If there was a way to make use of this free space, it would greatly reduce costs of operating and maintaining costly storage infrastructure and increase efficiency in the process as well. The threat of data exfiltration from organization computers where employees work is also a major concern. If an unauthorized person were to gain access to the computer, they would have access to all data in that device. So far organizations have spent millions of dollars annually and are still paying for Cloud Storage services. However, the security of the data that is stored on the cloud cannot be guaranteed. VAULT plans to address this problem in a decentralized methodology. VAULT will encrypt, segment and store data in multiple locations in a pooled storage space and use a blockchain to map hashes to actual locations. The blockchain will act as a Distributed Hash Table (DHT) in this scenario. The application will fragment and store file blocks throughout a peer to peer network in a redundant manner. This redundancy is achieved by implementing the Reed-Solomon Error Correcting codes. These blocks can then be used to recover the original file up to a maximum of N-2 peer failures (N – number of peers one file will be distributed) depending on the Reed-Solomon configuration.

II. LITERATURE REVIEW

Lampson, 1992 describes a methodology to perform authentication is distributed systems. The paper describes both a theory of authentication in distributed systems and a practical system based on the theory. The developed system executes commands as a result of remote procedure calls, and it handles public and shared key encryption. Furthermore, it explains several other security mechanisms, both currently existing and proposed [1].

Wicker and Bhargava, 1994 explains Reed-Solomon (RS) Codes and their use in contexts where burst errors are needed to be corrected. They also express their views on a wide range of applications in digital communications and data storage where Reed-Solomon codes are used. Furthermore, this paper describes and analyzes the performance and efficiency of Reed-Solomon Codes [2].

Researchers were able to identify similar products in the realm of distributed data storage. Of them, Ceph was identified to provide data safety for mission-critical applications. It provides virtually unlimited storage to file systems. Applications that use regular file systems can use CephFS with POSIX semantics without any integration while automatically balancing the file system to deliver maximum performance [3].

III. METHODOLOGY

A. Distributed Ledger

A single database that is consensually shared and synchronized across multiple institutions, geographies or on a smaller scale, nodes (computers) is called a Distributed Ledger. It eliminates the need of a central authority to remain responsible for the data and its security. Information stored in the distributed ledger / blockchain is securely and accurately stored with the help of proven cryptographic techniques used for hashing and sometimes encryption which can be accessed using keys and cryptographic signatures. The functionality of the distributed ledger in VAULT is that it provides an immutable record of files uploaded to VAULT from ‘time zero’. It also records actions taken on the file such as share or delete. This allows the application to track each individual file and its activity. The distributed ledger or blockchain inside VAULT will hereafter be called as “Fragchain”. Fragchain is a combination of two main and specific components.

1) Local Blockchain Logic

The logic governing the local chain is defined in this module. It exposes an API that is used to manipulate the local blockchain. VAULT’s Fragchain uses the Realm database for storage. It is a widely used key value mobile database which is compact and easy to use. It should be noted that the current data structure can be easily modified to be used for a different use case altogether. Fragchain was developed in a way which allows a developer to mutate the data structure without breaking blockchain specific functionality, giving the

developer the freedom to store any kind of data in the distributed ledger. This allows Fragchain to be used in scenarios where a distributed ledger is required.

a) Data Structure of Fragchain

Data that needed to be persisted for VAULT to function were mapped into a specific data structure that uniformly represent the fragmenting hierarchy. The data written to the blockchain is structured as depicted in Fig. 1.

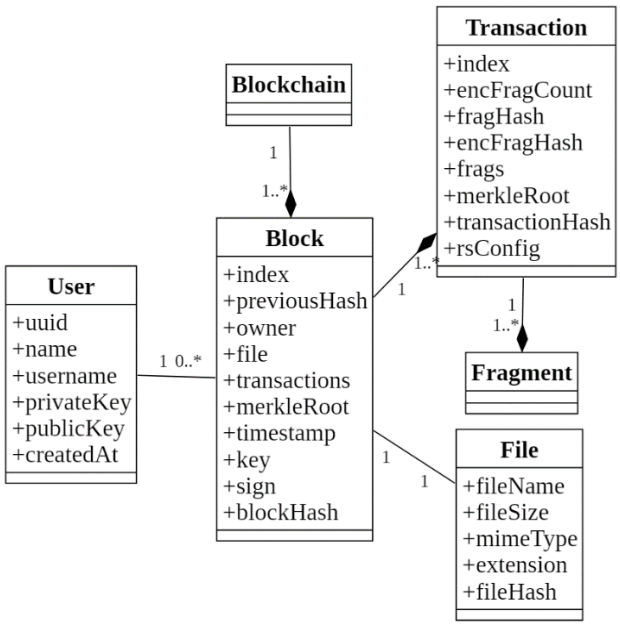

Fig. 1. Block Data Structure

Block contains a list of transactions which in turn contains a list of fragments. The figure below depicts the cardinalities of components of the data structure. The entire block is then saved on to the chain with references to the authenticated user and details concerning the original file. Once a file is uploaded, the file is chunked into pieces whose size depends on the host's capabilities. These chunks are then processed and fragmented further using Reed Solomon Erasure Coding. The chunks map to Transactions and the pieces generated from erasure coding are mapped into fragments. Therefore, a Block can contain one or more Transactions and a Transaction always contain 6-8 Fragments depending on the Reed Solomon Configuration [4].

b) Security of Fragchain

Three main components are addressed here when considering the security aspect of Fragchain.

INGEGRITY

A Merkle Root is calculated for Transactions and Fragments individually. This step ensures the integrity of data stored to an excessive degree. If a block is modified voluntarily or involuntarily, the validation mechanisms that verifies the hash tree would detect the abnormality and fail. Merkle Root calculations are carried out twice when a block is generated. Firstly, the fragments are hashed individually, and their hashes are used to calculate the FragmentMerkleRoot which is stored in the Transaction as 'merkleRoot'. Then the transactions are hashed individually, and the transaction hashes are used to generate the TransactionMerkleRoot which is subsequently stored in the Block as 'merkleRoot'. The Block's hash is then derived using parameters such as its index, previous hash, owner, timestamp and importantly Merkle Root. This single value signifies the integrity of Transactions and Fragments when aggregated.

ORIGIN AUTHENTICATION

Digital signing is a technique used to validate or verify the authenticity and integrity of data or information. It also provides the added assurance of evidence of origin and identity which prevents the problem of tampering and impersonation in digital communications. Fragchain uses digital signatures to sign individual blocks before they are added to the distributed ledger as shown in Fig. 2.

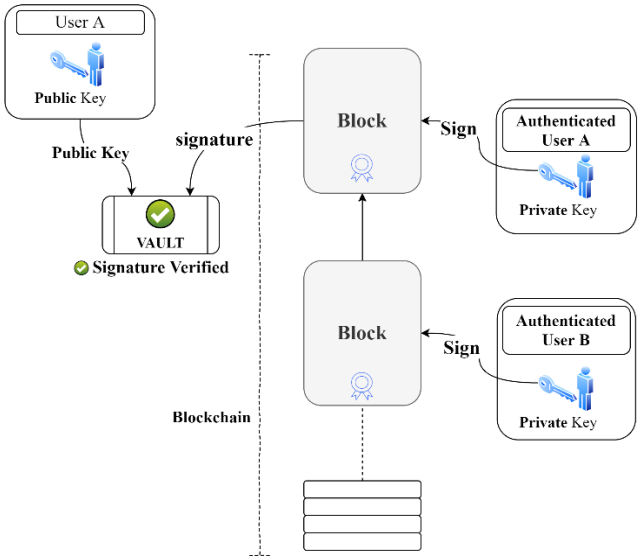

Fig. 2. Block Signing

DISTRIBUTED LOCKING TO SOLVE CONSENSUS

It is extremely important that the shared ledger is modified in a mutually exclusive manner. When multiple nodes try to perform modifications to the ledger, those modifications should be performed synchronously such that it isn't subject to mutations intermittently. To counter said circumstances, mutual exclusiveness was implemented in the form of a distributed lock when performing integrity changing operations such as modifications or additions on the ledger. A Distributed Lock Manager (DLM) was implemented in JavaScript (NodeJS) that emulates the DLM of Redis, Redlock [5] [6]. The following properties are achieved as minimum guarantees so that distributed locks are utilized in an effective way [6];

1. Safety – Mutual Exclusion. Only one participant / node can hold the lock at any given time.
2. Deadlock Free – It should eventually be always possible to procure a lock, albeit the node that had the lock crashes or fails.
3. Fault Tolerant – Nodes should be able to procure and release locks as long the majority of nodes are live.

These data structures, specifically the ones that hold file and fragment data contain fields that are designated to store file digests. When fragments are generated, Message Digest 5 (MD5) hashes are computed for each file and stored alongside the Fragment. These are then queried when a file is reconstructed to validate the integrity of the regenerated file. The underlying database is encrypted using AES-256 using a peer-key [7], which is configured by the administrator of the

swarm when configuring for the first time. In the event someone with physical access to the computer / device steals the raw database file, they won't be able to read the contents of it.

2) Messenger Module

Communication is a key factor in distributed systems to maintain state and propagate changes. The Messenger module of Fragchain works closely with the local blockchain logic to propagate changes made to the ledger to every node in the peer to peer network. The messenger module uses Socket.io to maintain real-time, bidirectional, event-based communications. The communication happens between a socket.io-client and a socket.io-server as shown in Fig. 3.

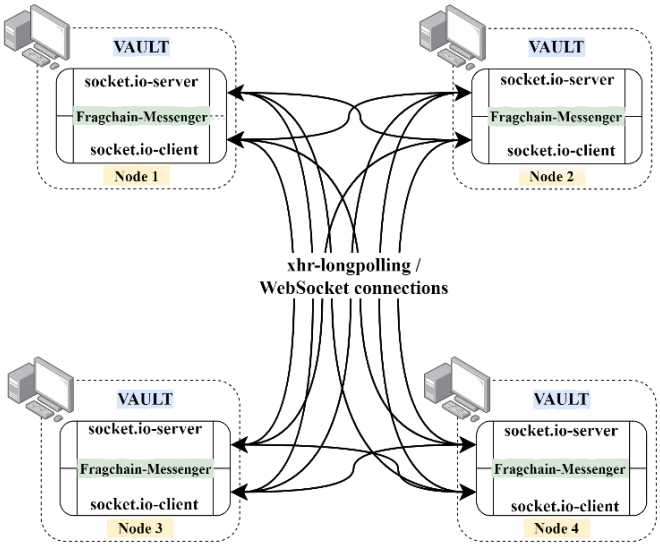

Fig. 3. Peer-to-peer connectivity

The event driven paradigm of socket.io lets the messenger module to respond to events that are sent or broadcasted over the peer to peer network. The events the Messenger module respond to are as follows.

1. SYNC_CHAIN_VERSION
2. SYNC_CHAIN_VERSION_RESPONSE
3. SYNC_CHAIN
4. SYNC_CHAIN_RESPONSE
5. BROADCAST_NEW_BLOCK
6. RECV_NEW_BLOCK

a) SYNC_CHAIN_VERSION

When Fragchain starts for the first time, or when it recovers from a crash, a SYNC_CHAIN_VERSION event is broadcasted. This event requests the chain version of all the peers in the peer to peer network. Upon receiving chain version responses from peers, the Messenger module checks the local chain version and compares it with the received version to determine synchrony. If the chain versions do not match, the entire blockchain is requested from a peer in the network which is then swapped with the local chain. The Messenger module internally calls the local blockchain API to replace the blockchain.

b) SYNC_CHAIN_VERSION_RESPONSE

Fragchain nodes that listens to SYNC_CHAIN_VERSION respond back with SYNC_CHAIN_VERSION_RESPONSE with the local chain version. The Messenger module locally calls the blockchain API to resolve the chain version which is then communicated to the requesting peer.

c) SYNC_CHAIN

When a peer receives a SYNC_CHAIN_VERSION_RESPONSE from a remote peer, the chain version is compared and the local chain is updated if the versions do not match. The SYNC_CHAIN event is emitted to the peer who responded with the higher chain version. The emitting node requests the blockchain of the listener so that is can be swapped with the obsolete chain.

d) SYNC_CHAIN_RESPONSE

Upon receiving SYNC_CHAIN event from a remote peer, the system queries the local chain and creates a stream-able object that contains the local blockchain. This is then streamed to the peer that requested the latest blockchain with SYNC_CHAIN_RESPONSE as the event name.

Once the SYNC_CHAIN_RESPONSE is received by the peer, it starts to asynchronously save the incoming blockchain data by swapping the outdated blockchain with the latest version.

e) BROADCAST_NEW_BLOCK

When a file is uploaded to Vault, Fragchain creates the block, and after acquiring the lock, the block is saved to the local chain. The BROADCAST_NEW_BLOCK event is then fired and block is then broadcasted to the entire swarm of peers.

f) RECV_NEW_BLOCK

Upon receiving a new Block from a remote peer, Fragchain tries to acquire the lock before saving the block to the local chain. When the lock is acquired, the received block is saved to the local chain after which the lock is released.

B. Redundancy Module

When a user uploads a file to VAULT it will be compressed, encrypted and then sent to the Redundancy module where Reed Solomon coding will be used to segment the file into shards in a redundant manner. The Redundancy module mainly achieves two goals.

1. Reduction of the amount of storage space needed for backups.
2. Increase in the assurance of data availability.

1) Redundancy Module - Encoder

The encoder takes a single file as an input and returns six coded-files as outputs. When a user uploads a file, it will be compressed and then encrypted initially. The encrypted file will be temporarily saved into the local node, and then the system will automatically invoke the encoder function. Encoder function reads the file and loads it into four different equally sized buffers as shown in Table I.

TABLE I. FILE (HEX): 00 01 02 03 04 05 06 07 08 09 10 0A 0B 0C 10 11 12 1A 1B AA

Data buffer 01	00	01	02	03	04
Data buffer 02	05	06	07	08	09
Data buffer 03	10	0A	0B	0C	10
Data buffer 04	11	12	1A	1B	AA

Equation (1) and Equation (2) is used to build the parity buffers. One byte from each data buffer with the same index will be selected at a time which is then applied into the two equations to generate two parity buffers byte by byte.

$$\text{Parity 01 - P1} = d1 + d2 + d3 + d4 \quad (1)$$

$$\text{Parity 02 - P2} = (2 \times d1) + (4 \times d2) + (d3 \times 6) + (8 \times d4) \quad (2)$$

When a byte is read in NodeJS, it is automatically converted into a decimal integer value between 0 to 255 by default. Assume d1, d2, d3, d4 all are equal to hexadecimal value 0xAA. In such a scenario, value of Parity 01 and Parity 02 will be greater than 255, which then cannot be converted into one hexadecimal value, because of that reason finite fields will be used in the program to map the output value into a value between 0 to 255. Since there are some values in the equation that are needed to be divided by even numbers, the JavaScript finite field function cannot map them in-between 0 to 255. In such cases, instead of mapping into a value between 0 to 255 mapping will be done between 0 to 256.

When output values are mapped between 0 to 256, value of 256 cannot be saved into the buffer, hence the program will save it to the buffer as 0 and maintain a Boolean array which keeps the status of each byte which will be added to the parity buffer by checking whether the byte is either 0 or 256. At the end, the bit stream will be appended to the end of the parity buffers. This is shown in Table II in regards to Table I data.

$$\text{Size of a parity shard} = \text{sizeOf (data shard)} * (1+1/8) \quad (3)$$

TABLE II. PARITY BUFFERS

	Hexadecimal values					Bit values				
Index	0	1	2	3	4	0	1	2	3	4
Parity buffer 01	38	35	46	50	199	0	0	0	0	0
Parity buffer 02	252	230	49	69	215	0	0	0	0	0

Once the encoding or else segmentation process is over, a total of six files will be distributed among the local network and the details such as digest values of each shard will be pushed to the blockchain.

2) Redundancy Module – Decoder

Functionality of the decoder is to revert the process performed by the encoder. It takes 4 files as an input and returns a single file which holds the same hash digest as the original file the user wants to download. When a user requests for a file stored within VAULT, details related to that file will be extracted from the blockchain in a secure manner. VAULT will then search for the six shards which were previously distributed among the network by the encoder. Once the system has found any four available shards from the network it will automatically invoke the decoder function. The decoder function checks the index value of each shard. Index values will be given to the shards when the encoding process happens based on the order the original file is segmented.

TABLE III. INDEX VALUES

Shard	Index value						
Data shard 01	***1***	00	01	02	03	04	
Data shard 02	***2***	05	06	07	08	09	
Data shard 03	***3***	10	0A	0B	0C	10	
Data shard 04	***4***	11	12	1A	1B	AA	
Parity shard 01	***5***	26	23	2E	32	C7	00
Parity shard 02	***6***	FC	E6	31	45	D7	00

Once the decoder function has the index value of the shards it will determine the number of available data shards and if all the four data shards are available, the decoder will concatenate the four shards together based on their index values and re-generate the original file right away. Refer Table III for the index values. If any data shards are missing, the available parity shards are used to generate the missing data shards byte by byte which is then stored inside a memory buffer. When regenerating the data shards with the help of parity shards, the bit stream which was appended to the end of the parity shard will be taken out and saved in a boolean array. When each byte generated, if the generated byte value equals to 0 the decoder logic decides whether it is actually a 0 or a 256, based on the boolean value in the array with the same index value as the parity byte. Equations in the tables below will be used when recreating the data shards from the available parity shards by the decoder. Further, each addition, division, multiplication or subtraction, finite fields will be used to map output values to a value between 0 to 255.

Assume one data shard is missing and Parity 01 is available. The recovery equations are shown in Table IV.

TABLE IV. RECOVERY EQUATIONS FOR PARITY 01

Missing data shard	Equation to be used
Data 01	D1 = P1 – D2 – D3 – D4
Data 02	D2 = P1 – D1 – D3 – D4
Data 03	D3 = P1 – D1 – D2 – D4
Data 04	D4 = P1 – D1 – D2 – D3

Assume one data shard is missing, and Parity 02 is available. The recovery equations are shown in Table V.

TABLE V. RECOVERY EQUATIONS FOR PARITY 02

Missing data shard	Equation to be used
Data 01	D1 = (P2 – 4xD2 – 6xD3 – 8xD4)/2
Data 02	D2 = (P2 – 2xD1 – 6xD3 – 8xD4)/4
Data 03	D3 = (P3 – 2xD1 – 4xD2 – 8xD4)/6
Data 04	D4 = (P2 – 2xD1 – 4xD2 – 6xD3)/8

Assume two data shards are missing and two parity shards are available. Resulting recovering equations are shown in Table VI.

TABLE VI. RECOVERY EQUATIONS WITH PARITY 01 AND PARITY 02

Missing data shard	Equations to be used
Data 01, 02	D1 = (4xP1 + 2xD3 + 4xD4 - P2)/2
	D2 = (P2 - 4xD3 - 6xD4 – 2xP1)/2
Data 01, 03	D1 = (6xP1 – 2xD2 + 2xD4 – P2)/4
	D3 = (P2 – 2xD2 – 6xD4 – 2xP1)/4
Data 01, 04	D1 = (8xP1 – 4xD2 – 2xD3 – P2)/6
	D4 = (P2 – 2xD2 – 4xD3 – 2xP1)/6
Data 02, 03	D2 = (6xP1 – 4xD1 + 2xD4 – P2)/2
	D3 = (P2 + 2xD1 – 4xD4 – 4xP1)/2
Data 02, 04	D2 = (8xP1 – 6xD1 – 2xD3 – P2)/4
	D4 = (P2 + 2xD1 – 2xD3 – 4xP1)/4
Data 03, 04	D3 = (8xP1 – 6xD1 – 4xD2 – P2)/2
	D4 = (P2 + 4xD1 + 2xD2 – 6xP1)/2

C. Encryption and User Authentication

A secure design has to be established in order to use VAULT since the application involves the use of a custom blockchain to store user information and furthermore the application offers a file sharing functionality. VAULT is designed and developed to comply with the CIA triad.

1) User Authentication

Each user will be registered with a user defined password and a username, and each user will be authenticated using a token-based JSON Web Token (JWT) for every action the user carries out. User details such as passwords and usernames are also stored in the blockchain itself and are stored as irreversible hash values making any attacks to gain access to the passwords futile.

2) Encryption

VAULT employs two Cryptographic principles which enables a cryptographically secure application namely Symmetric encryption and Public key cryptography. As soon as a file is chosen to be uploaded to the application, a Random key is generated using the crypto module, which is then used as the encryption key for the AES – 256 encryption algorithm when encrypting files. Since the crypto module generates only random bytes, the bytes are not predictable. The file is checked for the extension (mime) before being compressed and encrypted. And AES 256 in CBC mode is used as the encryption algorithm.

The main problem faced with this method is to share the secret key with the two parties and the key must be only known to each of those users. To avoid being vulnerable in an open channel, the random key that was generated will be stored securely in the blockchain using public key cryptography. As a user registers to the system, a public and a private key will be created. And the public key is later stored in the blockchain. This public key is used to encrypt the random key that was used to encrypt the uploaded file and later that encrypted random key is stored in the blockchain along with the file details. The Encryption and the decryption speed is based upon the key length and also the data size of the file to encrypt. Based on this input, the possible remediation to faster system runtime is to encrypt the Random key with the receivers' public key thus decreasing the time that is used to encrypt and decrypt a file. Summarizing, three steps are necessary to enable an index for private sharing:

1. Create Random Key to encrypt the compressed file
2. Create two key pairs (Public and private) and encrypting the generated Random key with the sender's public key.
3. Store the Encrypted Random key in the blockchain along with the file details.

Downloading the file works by applying the process reversely. At first the user will select the file that needs downloading. Then the user will be able to download the file using the index information stored in the blockchain. When the file is being downloaded, simultaneously the encrypted Random key will be downloaded. That key will be decrypted using the downloading parties' private key. Later on, that random key will be used to decrypt the downloaded file.

3) Share Functionality

A main component of VAULT is a decentralized version of a private file sharing functionality based on a Peer to Peer network.

In the scenario within the sharing function between two VAULT users, Alice and Bob. The user selects a file to be uploaded. As explained earlier in the cryptography section, a Random key will be used to encrypt the file. The file is then fragmented using the fragmentation module. Simultaneously the Random key that was used will be encrypted using the intended parties public key. Say if Alice needs to share the file with Bob, Alice encrypts the random key with Bobs public key. And if Alice wants to keep the file to herself, the Random key is encrypted with the Alice's Public key. That way, only Alice has access to the file. The encrypted Random key is then sent to the blockchain with the index details.

The download process is same as the upload process. Bob gets a notification that a file has been shared with Bob by Alice and then Bob can use Bobs' private key to decrypt the encrypted Random key. The bob will get the index which is publicly available in the blockchain and download the file. The file is then defragmented and uncompressed and then decrypted using the Random key.

D. Machine Learning for Distributed Storage

Storage management within VAULT provides the facility to create and manage storage space for users to store their data. Individual nodes will contribute storage space to the peer-to-peer network thus creating a virtual storage pool. This virtual storage pool will create by collecting physical HDD or SSD drives of remote nodes via the network.

A machine learning model trained to predict a node weight that reflects the suitability of an individual node to store file fragments is used to dynamically decide which of the nodes should be prioritized when distributing fragments. This will be explored further in to the Node Selection procedure.

1) Node Selection

Machine learning techniques aims to learn from data to solve problems such as prediction and estimation. Before distributing file fragments throughout the peer-to-peer network, all the nodes in the network will be analyzed and weighted. Six nodes will be selected based on the predicted weight from the in descending order of said weights. The storage module uses a multivariate regression algorithm within the general machine learning literature. The node selection process performed by 'ml-regression-multivariate-linear' JavaScript library. When a node is required to distribute a file among the network it collects information such as available storage space (pool size), free storage space, used storage space and health status (SMART metrics) from all the available nodes in the system. This information will be fed into the regression algorithm and weights will be assigned to each and every node available in the peer-to-peer network. Based on those weights, VAULT will decide which nodes are most suitable to hold the shards.

2) Node availability through messenger module

This module act as a call center between nodes. Each time when a file is fragmented into shards and distributed among the nodes or else when a file is regenerated from shards which was previously distributed among the nodes, the particular node where the owner or the download/upload requester triggers the VAULT API, in turn triggers the application to check whether the particular nodes are up and responsive. This module uses Socket.io to maintain real-time, bidirectional communication using a websocket based communication protocol.

IV. Results And Discussion

VAULT's Fragchain differs from other blockchain solutions available because it does not use a computationally extensive Proof of Work (PoW) calculation to achieve consensus [8]. Fragchain is a distributed ledger that who's main function is to store data securely. Therefore, the implementation of a Proof of Work function was redundant [9]. This allows Fragchain to converge and achieve consensus within a matter of seconds as shown in Table VII.

TABLE VII. Comparison between existing distributed ledgers and VAULT's Fragchain

Blockchain	Time taken to converge
Bitcoin	~10 minutes
Ethereum	~ 12 seconds
Fragchain	~2 seconds

Furthermore, Fragchain does not record or contain alternate histories (ancient states) where the blockchain is branched and left stale. It has a single linear history that starts from the genesis block and spans forward in time. This has allowed Fragchain to be compact thereby utilizing the bare minimum storage required to store the blockchain locally. Fragchain automatically recovers from a hard fail and synchronizes the blockchain to the swarm default version when recovering. The messenger client of all peers in the swarm polls the crashed servers until it comes back up. This allows the swarm to self-heal without 3rd party intervention.

The advantage of using parity shards is that even when two shards are unavailable from the total of six shards, regardless of the index value of the shard, the original file can be regenerated. The use of Reed Solomon coding-based approach to segment the files has its perks. One of those perks is that it will reduce the storage space required for backups.

In regular file systems storage space required for backups equals to twice the size of the original file. But in VAULT, the storage space required for backups = 9 x (Size of the original file / 16).

TABLE VIII. STORAGE USAGE OF A REGULAR FILE SYSTEM

Regular system		
Backup space	2 x 8000	16 000 MB
Total Space	3 x 8000	24 000 MB

TABLE IX. STORAGE USAGE OF VAULT

VAULT		
Backup space	9 x (8000 / 16)	4500 MB
Total space	8000 + 9 x (8000/16)	12 500 MB

Approximately the total amount of storage space required by the VAULT to recover from a two-node failure situation is half of the size that will be required by the regular file systems as depicted in Table VIII and IX when a 8GB file is considered.

Fig. 4 depicts the availability of files uploaded to a VAULT swarm of 12 nodes or computers under a chaos test. The chaos testing on this scenario was carried out after uploading 10 variable sized files ranging from 100 MB to 500 MB of types .mp4, .zip, .img. Nodes were randomly shutdown and availability of files were queried and graphed. As shown in Fig. 4, 100% availability was achieved when 10/12 nodes were active confirming the N-2 redundancy metric.

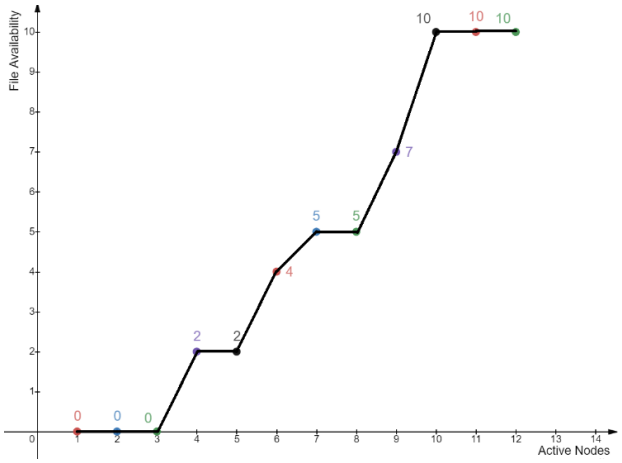

Fig. 4. Active nodes vs File Availability

V. CONCLUSION

It is evident from the functions and features of VAULT illustrated above that it successfully achieves and outperforms current applications in the niche market of local storage solutions. Since privacy is inbuilt into the application environment, it is ideal to be used in privacy critical organizations. The fact that VAULT is intended to run on existing general-purpose hardware makes it even more appealing to corporate environments with hundreds of workstations allowing them to save money on costly storage infrastructure. The decentralized nature of VAULT inherently eliminates a single point of failure thus defaulting to a much more resilient design. Intelligent resource allocation achieved via Machine Learning optimizations allow VAULT to perform file distribution much more effectively resulting in fewer reconstructions of files and even fewer files becoming unavailable.

REFERENCES

[1] M. A. M. B. E. W. BUTLER LAMPSON, "Authentication in Distributed Systems: Theory and Practice," ACM Trans. Computer Systems, vol. 10, no. 4 , pp. 265-310, (Nov. 1992).

[2] S. B. Wicker and V. K. Bhargava, "An Introduction to ReedSolomon Codes," 1994 .[Online]. Available: http://ieeexplore.ieee.org/book/0780353919.excerpt.pdf.

[3] Ceph, "ceph.com," Ceph, 03 2019. [Online]. Available: https://ceph.com/ceph-storage/file-system/. [Accessed 01 03 2019].

[4] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," Journal of the Society for Industrial and Applied Mathematics, 1960.

[5] M. Kleppmann, "How to do distributed locking," 06 Feb 2016. [Online]. Available: https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html. [Accessed July 2019].

[6] antirez.com, "Is Redlock safe?," 2017. [Online]. Available: http://antirez.com/news/101. [Accessed August 2019].

[7] realm.io, "Realm Database," [Online]. Available: https://realm.io/products/realm-database. [Accessed July 2019].

[8] BlockSplain, "Does blockchain size matter?," November 2018. [Online]. Available: https://blocksplain.com/2018/02/22/blockchain-size/. [Accessed July 2019].

[9] bitcoin.it, "Proof of work," bitcoin.it, 2018. [Online]. Available: https://en.bitcoin.it/wiki/Proof_of_work. [Accessed 2019].