# PP-FileShare: A Peer-to-Peer File Sharing Platform

Blockchains and Cryptocurrencies 601.441/641

**Xiaoyan Li**
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
xli152@jhu.edu

**Tan Nguyen**
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
tan_nguyen@jhu.edu

May 4, 2019

**Abstract**

We present PP-FileShare, a peer-to-peer file sharing platform. While we recognize the existence of IPFS and Filecoin, we also realize that the scale of these systems can be unfit for subsets of users. Therefore, PP-FileShare offers a minimal viable solution for small-to-medium-size system with additional capabilities. While we adopt features from existing systems, we also introduce a simplified challenge scheme with a modified Proof-of-Stake, data encryption, and other extensions for different applications using this architecture. We provide an implementation of our system with AWS EC2 instances running TCP connection, implemented in Python.

***K*eywords** File Sharing · Blockchain · Peer-to-Peer

## 1 Introduction

Since the emergence of Bitcoin and other cryptocurrencies which prove the utility and effectiveness of peer-to-peer distributed network, many start to apply blockchain technology to various applications. In this paper, we introduce another blockchain's application: PP-FileShare, a Peer-to-Peer File Sharing Platforms that incentivizes users to share and store files in an efficient and secured manner. While we recognize the existence of similar systems like IPFS, Filecoin, and BitTorrent, we want to introduce a framework, or a minimal viable platform, for systems at smaller scale to adopt. We will also introduce different new extensions for the platform such as file encryption and simpler challenge scheme that serves a greater purpose for a file sharing system.

The paper is organized as follows. In section 2, we will discuss provides background on existing work like IPFS and Filecoin and an efficient public key traitor tracing scheme, which will be used as PP-FileShare's data encryption scheme. In section 3, we will discuss our construction of PP-FileShare as minimal viable platform with in-depth analysis on each component. In section 4, we will analyze in depth different possible attacks on our system with potential solutions for those attacks. In section 5, we offer some sample applications and evaluate how our overall system, especially in the context with other systems like BitTorrent, the incentive scheme and its comparison to existing centralized systems.

## 2 Background and Related Work

This section will discuss existing works regarding distributed file sharing system and other background works that PP-FileShare draws inspirations on.

### 2.1 IPFS and FileCoin

InterPlanetary File System (IPFS)[5], developed by Protocol Labs, is an effort to replace HTTP protocol. While HTTP protocol downloads each file from a single computer, which is slow due to limited bandwidth and unreliability, IPFS claims to possibly save the bandwidth cost by 60%. IPFS achieves this by implementing content addressing, instead of IP addressing like HTTP for each packet in the system. The system splits the content of the file into smaller chunks, builds a Merkle tree, and computes the address for the file as the Merkle root hash. From the hash function's properties, it guarantees that this address is unique for each file uploaded to the system. IPFS also implemented version control, which uses Merkle directed acyclic graph (DAG), a data structure closely resembling the architecture of Git system.

Filecoin [6] introduces a cryptocurrency built on top of IPFS. By doing so, Filecoin provides a market for buyers, who is willing to rent out spaces, and sellers, who pays to get their files distributed in the system. Filecoin introduces incentives to IPFS system. However, the main contribution of FileCoin is its challenge scheme, Proof of Replication. Since space plays a more important role in the system, Filecoin stirs away from using traditional Proof of Work (in Bitcoin and Ethereum) or Proof of Stake (like in PeerCoin). Instead, FileCoin uses Proof of Spacetime and Proof of Replication[2]. Proof of Replication is a challenge scheme employed by FileCoin that ensures the $n$ copies of files claimed to be stored by server $\mathcal{P}$ is actually stored, detecting Outsource attacks and Sybil attacks. In both cases, the data is only stored once as opposed to $N$ times claimed by the servers. This is proved through a ReplicationGame, in which if an honest verifier $\mathcal{V}$ is given $N+1$ files, and an adversary $\mathcal{A}$ only storing $N$ files cannot be verified, then the scheme is secure. Each file $\mathcal{F}$ is encoded with a reversible, distinct encoding - $\forall i, j, encode(F_i) \neq encode(F_j)$ - and committed to store such encoding. However, instead of PoRep, the PP-FileShare system uses the challenge scheme inspired by Proof-of-Replication, which will be further discussed in section 2.2.

## 2.2 Proof of Retrivability

Proof of retrievability[4] is a challenge scheme proposed to ensure that the data storage facility is storing the complete, untampered version of the client's data, verifiable in an efficient and provably secure way. The protocol works as follows.

The Jules-Kaliski scheme for PoRet pre-computes a limited amount of challenges. That is, the encoded file $\mathcal{F}$ is broken in to $n$ blocks with each block $m_1, ...m_n \in \mathbf{Z}_p$ for larger. The individual blocks can be authenticated by user. In order to do this, user chooses pseudorandom function generated key $k$ and a random value $\alpha \in \mathbf{Z}_p$ which serves as their secret key to calculate an authenticator $\sigma$ with. The verifier then chooses random challenge of index-coefficient pairs, which the prover calculates responses with. The response is then compared to the stored authenticator.

## 2.3 Efficient Public Key Traitor Tracing Scheme

Efficient Public Key Traitor Tracing Scheme [3] is a public key encryption scheme where there is one encryption key, but *many* private decryption keys. Key features of this scheme includes:

- Each legitimate user can decrypt the message using its own key
- If a new decryption key was created by a collusion of multiple users, it is efficient to trace out all the "traitors".

Here, we want to provide an overview regarding the key generation, encryption, decrytion, and tracing scheme:

- Key Generation: Output a public encryption key $e$ and a list of private decryption keys. The scheme uses a public *linear space tracing code* $\tau$, corresponding to, but not revealing, $l$ private keys. The scheme generates a public key and $l$ private keys using a variation of Decision Diffie-Hellman problem.
- Encryption: Encrypt message M with a random element in $\mathbb{Z}_q$, similar to the Diffie-Hellman scheme
- Decryption: Decrypt the message using the corresponding codeword in $\tau$ and its private key
- Traitor Tracing Scheme: Due to the key generation scheme, a collusion of private keys will map to a multiple of codewords in $\tau$. Therefore, using dynamic programming to generate all possible codewords, we can trace back to the collusion of private keys.

The original paper provides a detailed proof on accuracy of encryption and decryption scheme as well as analysis on security of the system (mainly on the difficulty of discrete log problem) and on the time of traitor tracing (which is $O(l^2)$ in naive implementation with $l$ be the number of private keys generated). This efficient algorithm discourages the private key owners from leaking their keys or joining their keys to generate new private keys.

## 3 PP-FileShare: Peer-to-Peer File Sharing Platform

Since the existing IPFS and Filecoin systems have ambitious goals of "rebuilding the Internet" and creating decentralized storage market, many distributed systems can benefit greatly from similar ideas while not completely depending or being built on top of the current system. More importantly, IPFS doesn't provide data encryption for security. This is why we want to propose a PP-FileShare, a minimal viable peer-to-peer file sharing platform.

For our contribution , we incorporate an array of ideas needed to build this minimal viable platform such as file swapping, version control, public key traitor tracing scheme, and verification system. Our contribution of PP-FileShare consisted of simplifying and integrating existing systems into one minimal, efficiently working system, ready to deploy for multiple platforms and applications.

## 3.1 Protocol

We present our complete system in **Protocol 1**. For our system, each user is considered to be a whole node with an address, public key, and private key. Each user will have a certain amount of storage space $s$ in the system that they can upload files and/or rent out.

**Protocol 1** PP-FileShare

1. **Setup**
   - $INPUTS$ : address $addr$
   - $OUTPUTS$ : None

   (a) Generate $pub\_key$ and $secret\_key$
   (b) Specify $s$ as storage space willing to rent
   (c) Connect to network with $addr$
2. **Upload File**
   - $INPUTS$ : file $F$
   - $OUTPUTS$ : public hash $pub\_hash$

   (a) Split $F$ into $n$ smaller files, each with $m$ predetermined bytes
   (b) Compute the Merkle hash for $F$ as $pub\_hash$
   (c) Return $pub\_hash$
3. **Upload 1-to-1 Encrypted File**
   - $INPUTS$ : file $F$, public key $pub\_key$
   - $OUTPUTS$ : public hash $pub\_hash$

   (a) Encrypt $F$ with $pub\_key$ as $F\_hash$
   (b) Run $UploadFile$ on $F\_hash$
   (c) Return $pub\_hash$
4. **Upload 1-to-n Encrypted File**
   - $INPUTS$ : file $F$
   - $OUTPUTS$ : public hash $pub\_hash$

   (a) Generate a public key $pub\_key$ and $l$ private key $pk_1, ...pk_l$ using public key traitor tracing encryption scheme
   (b) Run $Upload1 - to - 1EncryptedFile$ on $F$ and $pub\_key$
   (c) Return $pub\_hash$
5. **Distribute Upload**
   - $INPUTS$ : address $receiver\_addr$, file $F$, public hash $pub\_hash$
   - $OUTPUTS$ : $SUCCESS$ or $FAILURE$
   - $ASSUMPTION$ : Party $\beta$ with address $receiver\_addr$ has agreed to make connection and ready to receive file $F$

   (a) Transfer $F$ to address $receiver\_addr$
   (b) For interval within time:
   (c) run $VERIFY$ with $B$ to prove $B$ store $F$
   (d) If $VERIFY$ succeeds and interval = last, return $SUCCEESS$. Else, return $FAILURE$
6. **Verify**
   - $INPUTS$ : public hash $pub\_hash$, node address $addr$
   - $OUTPUTS$ : $SUCCESS$ or $FAILURE$

   (a) Send to $addr$ challenge $C(pub\_hash)$
   (b) Receive $addr$ solution $S$
   (c) Return check($S$, $pub\_hash$)
7. **Request**
   - $INPUTS$ : publish hash $file\_hash$,
   - $OUTPUTS$ : File $F$ or $NONE$

   (a) Create $contract$, deposit reward $v$ and embed $file\_hash$ and requester $addr$
   (b) Distribute to neighbors
   (c) verify($\beta.file\_hash, \beta.addr$) for Proposer $\beta$
   (d) **If** $SUCCESS$**:** $\beta$ receives reward $v$, retrieve $F$
   (e) Return $F$
8. **Update Files**
   - $INPUTS$ : File $F_1$, old publish hash $old\_pub\_hash$
   - $OUTPUTS$ : public hash $pub\_hash$

   (a) Initialize a pointer $p$ for $F_1$ pointing at $old\_pub\_hash$
   (b) Run $UploadFile$ on $F_1$

### 3.2 Challenge scheme

#### 3.2.1 Modified Proof of Stake

Because PP-FileShare drives node participation with monetary incentive, it is crucial that the transactions are documented and stored on the blockchain. However, unlike conventional Proof of Stake [7] in which the miners participate with monetary stake, in the modified scheme, we use the individual miners' average size of files stored in the last M blocks in the chain. The total size of files are confirmation by Proof of Retrievability mentioned below. Each node needs to spend resources to rent out spaces to store files for other nodes; therefore, their corresponding stakes are higher in this system. In addition, the system needs to choose N blocks ahead of time to prevent stake grinding and nothing at stake attack in regular Proof of Stake, as well as collusion attack mentioned later in Section $4.2$. We also propose several alternative stake options to the system. One option for stake can be the amount of spaces a node's distributed files take up in the system. When a node distributes files, it needs to pay to other nodes as rewards; therefore, the amount of stakes will be proportional to the amount the node invests in the system. Another alternative option can just be the traditional implementation of proof of stakes, though it is not recommended as monetary value is less important in our system.

#### 3.2.2 Proof of Retrievability

The challenge scheme in this distributed file sharing system is different because the node's storage availability and the files that they claim to have stored are primary concerns of the system. The first concern of claimed available storage is not dealt with in this minimal platform, but possible solutions including Proof-of-Storage are proposed by other distributed file sharing platform constructions such as Chia. In this MVP we mainly focus on the file storage, along with ensuring its validity and integrity. This is achieved by issuing modified Proof-of-retrievability challenges.

**Set-up:**

The file uploader splits file $\mathcal{F}$ with $size(\mathcal{F}) = k$ into $n$ pieces equal size of $\frac{k}{n}$. Then, the pieces are used as leaves to generate a binary Merkle tree with Merkle root $r$. The entire tree with its leaves gets uploaded into the network, as described in **Algorithm 1**. $\mathcal{N}$ challenges are pre-computed by the uploading node. A subset of the challenges is distributed to receiver nodes along with decryption key (see section $2.3$).

**POS procedure:**

To require the storage nodes to prove that they have the correct file, the requester sends challenge $c = (b, m)$ to the challenger where $b$ is the starting bit position in the original file $\mathcal{F}$ and $m$ is the subset of sequential bits starting at position $b$. The challenger returns the Merkle Proof $\mathcal{P}$ of the block containing the bit $b$. The challenge is considered complete if all of the following evaluates to true.

- $leaf(\mathcal{P})[b]$ = val($b$) at $\mathcal{F}$
- $leaf(\mathcal{P})[b{:}b + len(m)]$ = $m$
- verify($\mathcal{P}$)

**Correctness:**

If the hash function generating the Merkle tree selected to be pre-image resistance and collision resistance (our implementation uses $SHA256$ which is believed to be an instance of such), then its tamper resistance and collision resistance property follows that proved previously [1]. Following this, the verifier need not have copy of the entire file to verify $\mathcal{P}$. It only need to check the Merkle root can be validly generated from $\mathcal{P}$. Since this is the most time consuming part of the verification, the verifier will dominate the run time. $\mathcal{P}$ has $log_2(n)$ levels, which is then number of hashes it needs to compute. Therefore, the run time is also $O(log_2(n))$

The possibility of forming another block with the same $m$ bits starting at position $b$ is $\frac{1}{2^{len(m)}}$ given the file is encoded in binary. However, this probability is also multiplied with the probability of fabricating a block of different value with a valid Merkle Proof that evaluates to the same Merkle root, which the above had cited to be difficult. Therefore, the position and chunk verification is merely to prevent the same Merkle Proof from being reused, and the length of $m$ is not crucial to verification correctness, although a greater number of bits is recommended.

### 3.3 Usage of Modified PoRet

For secure information exchange, this architecture uses smart contracts, which is simulated but unfortunately cannot be implemented with security in the Python framework the authors chose. The following sections outline the ideal implementation of PP-FileShare.

**Algorithm 1** Verifier in Proof of Retrievability

```
procedure CHECKVALIDITY(proof, root)
    parent ← parent of leaf
    h ← hash(parent.leftChild, parent.rightChild)

    while parent is not $None$ do
        temp = parent
        parent = parent.parent
        other = parent.child that is not temp
        h = hash(h, other.hash)
    other = parent.child that is not h
    return hash(h, other)=root
```

### 3.3.1 Upload File

As Section 3.3.2 indicates, the data in upload contract is the Merkle tree generated from the file. In order for the uploader to verify that the file is stored for the entire time duration - implicitly computed from the total amount of reward and the value of incremental reward, a lightening-channel-like micropayment channel is set up between the uploader and the storager. After every time increment, the uploader sends a different challenge $c$ to the storager, and if the POS procedure in Section 3.3.2 evaluates to true, then both parties sign and create a transactin with a larger sequence number. If the storager fails the proof, then after the time lock the uploader can create and publish a transaction with larger sequence number and get refunded the remaining award.

Note that once the uploader uploads the file from local storage, they no longer need to store the original file but only the challenges $c$. This is $numbits(b) * num(c)$ bits.

**Algorithm 2** Distribute File

```
reward = 0
timelock = some arbituary future time
author, pubkey = None
prev = None

procedure CONSTRUCTOR(value, author, pubkey, prev)
    author = author
    pubkey = pubkey
    prev = prev
    if value > 0 then
        reward = value
    else
        $ERROR$

procedure DEPOSIT(value)
    request.reward += value

procedure BECOMESTORAGE(storager)
    open up microchannel

    with every PoRet completed,
    pay increment amount

procedure WITHDRAW(withdrawer, amount)
    if withdrawer = author then
        transfer(amount) to proposer

procedure REFUND(now)
    if withdrawer = author then
        transfer(amount) to proposer
```

### 3.3.2 Request and Download File

**Procedure:**

In order for the requester to pay the proposer, the proposer to receive the correct amount of payment, and the requester to receive the correct, untampered file that they requested, another request smart contract is created and called upon. The requester deposits the proposed amount of reward in a contract with the designated withdrawer as the requester itself and a proposed time-lock. When a proposer proposes to have the file $\mathcal{F}$, the requester sends challenge $c$ to the proposer. The proposer returns their proof $\mathcal{P}$ generated from $c$ along with the data in the contract. If the $\mathcal{P}$ is verified, then the data is stored in the contract and can be retrieved only if the designated withdrawer gets changed to the proposer. The requester can then retrieve their requested data, and the storage node can reclaim their reward. If they didn't reclaim their reward within a given time, then the reward is refunded to requester. If the requester didn't approve of the storage node's contributed file within the given time, they are refunded and do not claim the file.

**Analysis:**

Given the time-lock in the contract, if the file request is never responded to by any node in the system - in case of file not stored or DOS attacks - the requester will be refunded their reward.

The pre-designated withdrawer is yet another attempt to prevent intermediary nodes from claiming the deposit. Once a proposer responds to the request, the requester sends them the challenge. Note that this requester doesn't have to be the uploader of the file. As in the encryption scheme decryption keys are sent to designated retrievers, the challenges $c$ can be sent in a similar fashion.

The data can only be retrieved after the withdrawer is assigned to prevent the requester from denying the reward to the proposer. Afterwards, it is removed from the contract to save storage when creating blocks and for security purposes.

**Algorithm 3** RequestFile

```
request = {requester, hash, reward, proposer, timelock, pubkey, data}
withdrawer = None
timelock = some future time

procedure CONSTRUCTOR(hash, reward, requester)
    request.requester = requester
    request.hash = hash
    request.proposer = requester
    withdrawer = request.proposer
    if reward > 0 then
        request.reward = reward
    else
        ERROR

procedure DEPOSIT(value)
    request.reward += value

procedure SET PROPOSER(proposer, signature)

this is called after proposer and requester establish private connection and the proposer completes successfully the requester's PoR challenge

    if verify(request.pubkey, signature) AND withdrawer == request.proposer then
        withdrawer = proposer

procedure PROPOSE DATA(data, proposer)
    if hash(data) = request.data AND proposer = request.proposer then
        request.data = data

procedure DOWNLOADDATA(requester)
    if request.data != None AND withdrawer != request.requester then
        data = request.data
        remove request.data
        return data

procedure WITHDRAW(proposer, amount)
    if proposer = request.proposer then
        transfer(amount) to proposer
```

### 3.4 Version Control

As with current file storing systems, the data stored within is constantly updated. To accommodate for these changes, this system provides version control capabilities. Version control is implemented using digital signature, which is based on the RSA-encryption scheme. In order to maintain uploader identity, users sign the Merkle Root of file $\mathcal{F}$ prior to uploading and embed their public key with their signature into the contract. When the original uploader wants to update the file, they can sign the new file, $\mathcal{F}'$'s Merkle Root, include the Merkle Root of $\mathcal{F}$, and upload the file as outlined in the Upload protocol in Section 3.3.1. The storager nodes can check whether the two uploads are indeed by the same user by verifying the signature using their stored public key from the previous upload. Given the secured digital signature scheme, we believe that files uploaded to our system cannot be replaceds by other malicious nodes in the system.

### 3.5 Data Encryption

Due to the decentralization property of our platform, uploaded files can be viewed by any node that chooses to store the file. Therefore, if it is not encrypted prior to being distributed, malicious parties could gain access to the information. We solve this problem by using the public encryption traitor tracing scheme described in section 2.3, especially for files needed to decrypted by multiple nodes. For file needed to be decrypted by only one node in the system, storager node can simply encrypt the file with the user's public key. The node, therefore, can decrypt with its own private key. For the case of multiple nodes decrypting a file, to secure the file, we generate a public key and $l$ distinct private keys, encrypt the file with the public key. When the uploader sends the public hash of the file to the receiver (in a separate channel), they can also send a decryption key. While this can pose a security issue should the party holding the secret key chooses to leak the key, but we reasonably believe the party has no incentive to do so, providing the possible applications using our system. Furthermore, although parties receiving the private keys could collude and combined their decryption keys together to form a new decryption key (given our key generation scheme as a discrete log problem), the traitor tracing algorithm mentioned above allows the uploader to discover which keys were used to generate a new key efficiently. Therefore, the uploader can broadcast this bad behavior to the network and potentially blacklist the colluded receivers.

## 4 Potential Attacks

### 4.1 Man in The Middle attack

Currently in the upload protocol, the number of challenges (defined in section 3.2.2) are pre-computed and used for PoRet. The challenges are distributed along with the decryption keys to requester from the upload. In theory, the storager would not know the challenge, and the randomness in the challenge ensures the storager has the entire file. However, note the challenges doesn't have to be completed. Therefore, if the storager colludes with another node $q$, then $q$ can respond to a variety of requests for a single file $\mathcal{F}$. Since $q$ doesn't store $\mathcal{F}$, it cannot complete the challenge and the contract either expires or stays open. However, in responding to the request, $q$ receives a sequence of responds $c_1, ..., c_n$, which it can leak to the storager node. Therefore, in order to pass PoRet, the storager node no longer has to store the entire file. This can be solved by including a time-dependent nonce suggested by the original PoRet protocol, which requires the challenges to be publicly verifiable and generated by the requester instead of sent to by the uploader.

### 4.2 Collusion

An inherent problem for our current system is that the file size within an exchange is determined by the proposer and the storager. In the exchange block, there is no mechanism to verify if the size of the file is accurate since the data is not recorded on the chain; therefore, if two parties collude, they can inaccurately report the file size of the exchange, allowing both parties to have much higher stakes in the system. We proposed some solutions for this attack. As mentioned above, we choose the block verifier M blocks ahead, so the collusion will not yield immediate effects and its influence weakened. Other nodes can challenge this bad exchange (based on off exchange rate \$/bytes*seconds) and refuse to accept it to the block. Another solution is to have a fixed exchange file size where all the exchanges in the system is the same; therefore, we can check collusion through invalid exchange if the reward is too low.

### 4.3 Deny of Services From Intermediary Nodes

Immediate neighbors of requester nodes can deny to pass uploading and requesting contrasts from a node. The difficulty of this depends on the number of neighbors $N$ the node has, and administering it would mean taking over (physically or by incentive) $N$ nodes. To solve physical control of the $N$ nodes, the DOS-ed node can create connection with other nodes on the network. If the control is by incentive - the neighboring nodes are bribed to DOS, then file-partition-storage can be introduced as incentive on the intermediary nodes. That is, instead of the entire Merkle tree, which is necessary to generate the proof, the file can be broken into smaller sub-chunks with their respective Merkle Tree and Root. The intermediary nodes can then store the sub-chunks for reward as well, which reduces the possibility of bribery as their own reward is at stake.

### 4.4 Deny of Services From Storage Nodes

Another DOS could be from the storage nodes. That is, the entire network refuses to store the uploader's file. However, this is difficult- either requires taking over the entire network or bribing all the nodes in the network- which consumes either much computing power or much monetary asset. Therefore, we think this attack is unlikely.

## 5 Applications and Evaluation

### 5.1 Implementation

In addition to this paper, we also include a short implementation of our system (around 1700 lines): `https://github.com/xli98/PP-FileShare`. The system is intended to be minimal, but still included all key features such as exchange scheme, Merkle proof, challenge and response scheme, etc. We did not implement data encryption scheme for this version.

### 5.2 Applications

There are many possible applications that can use our PP-FileShare platform to be more secure and efficient, like distributed job application platform, machine learning file sharing system, medical data file sharing, etc. Here, we will go in depth of how to build such system for distributed job application platform while providing some example extensions that a platform like machine learning file sharing system can utilize.

#### 5.2.1 Distributed Job Application Platform

We want to propose a distributed job application platform where each company and potential applicant will set up a node in this distributed network. Each applicant will upload their encrypted applications and distribute their file to the system with monetary rewards. When the applicant wants to apply, they will submit the applicants' Merkle root hash and the decryption key in a separate channel. The company then requests the application from the network, decrypts, and retrieves it. If applicants want to send a customized application to a company, they can encrypt their applications with the company's public key. Applicants, and sometimes companies, can gain monetary benefits from renting out spaces to store others' applications and send them when requested. The network is self-sufficient as long as participating nodes provide enough storage space.

The network can operate well under the assumption that each node acts along its incentive. A company will not leak applications after decryption as they do not want to lose their talents. Data in this case will be accessed by the applicants and the companies only, rather than being used by a third centralized party. The chance of losing data is low since data is being copied across the network (given sufficient rewards). However, this system has weaknesses that need to be addressed with further extensions. The network will know of the smart contracts; therefore, it can associate the public key with address and file hash. In this system, since it is unnecessary to use version control (applications of the same applicants should not be connected), a possible solution is to create a group similar to an anonymity set among different trusted applicants where they may share the same public key and disassociate the file with the owner. Another possible difficulty with our system is distributing 1-to-1 customized applications. Since only one company can decrypt this file, it is wasteful and difficult to attach multiple copies across the network with high rewards.

#### 5.2.2 Machine Learning File Sharing System

In the age of big data and machine learning, it is especially inefficient to store huge files of data multiple times. Therefore, we propose that research groups can use our proposed platform to form a data sharing distributed system. An extension that we want to propose is to allow data to be read and processed over the network instead of retrieving the complete set of data. An alternative that the system of this scale can do is to store the same data in multiple nodes and allowing data access to be parallel across multiple GPUs. This process will lower the file I/O bottleneck that many systems suffer when reading from a single memory.

### 5.3 Evaluation

#### 5.3.1 Incentive

As nodes are merely storage hardware owned by users in the network, there needs to exist incentive for nodes to willingly participate in the network. Therefore, the monetary incentive system is introduced. Its benefits, aside from encouraging encouraging participation and presenting DOS attacks in uploading and requesting files, also includes accelerating transaction as per seen in BitCoin, contracts with larger monetary reward is likely to be processed faster.

#### 5.3.2 PP-FileShare vs. BitTorrent

One of the first peer-to-peer file sharing system, BitTorrent, allows users to share files voluntarily among another with tit-for-tat method. While many have suggested BitTorrent to be the go-to technology for private file sharing, it has many issues that our implementation of PP-FileShare can solve. The biggest concern with BitTorrent is that it lacks incentives. When a system is built based solely on volunteer, nodes are not willing to share files. Unless there is a high demand, files will get lost regardless. While BitTorrent allows downloads from multiple sources, it is not content addressing. This also means any source can change parts of the files, leading to a lot of security concern that BitTorrent is known for. PP-FileShare allows verification through the Merkle tree hash, which is also hard to fabricate without having the real file. Finally, PP-FileShare has secured version control, which reserve modification right to only the file owners, and allow public verification for this as well.

#### 5.3.3 Comparison with centralized data storage

In comparison to the current centralized file storage system, this distributed system brings simplicity in file sharing. If a file is requested by multiple parties, then the user no longer need to upload individually to the parties or the respective centralized entities. Instead, the parties can request from the system. The distributed system also represents social implications. When translated to its application such as job application sharing, having a centralized entity holding private information bring different social connotations and is less trusted as opposed to storing in distributed systems.

Disadvantages of the system includes potentially increased length of time before file is stored in or retrieved from the system, which is dependent on the node activity in the system as well as the monetary incentive associated with requests. There could also be node attacks. If the file is large, the net storage price grows proportionally, driven by the market. However, considering that even in centralized system storage size is correspondent to storage space and arbiturarily set by the centralized party (consider Dropbox premium to free version), the cost incentive shouldn't be a great deterrent.

## 6 Conclusion and Future Work

We have proposed PP-FileShare, a peer-to-peer file sharing platform that is minimal and ready to deploy to many applications. We started with existing systems such as IPFS and Filecoin, which provides a good framework, but too complicated yet inadequate system for small-to-medium-sized systems to adopt. We simplifies these framework by introducing new challenge scheme with proof of retrievability and a modified proof of stake while incorporating data encryption scheme. We also implemented a sample of our systems in Python, using AWS EC2 instances with TCP connection. We offered some applications that can improve with further extensions from the minimal platform.

While PP-FileShare is ready to be deployed, there are many improvements we wish to make to the system. Regarding security, we hope to extend encryption scheme without limited private keys, so files can be shared to arbitrary number of nodes in the system. We also want to allow nodes to share parts of files to the system, so nodes, especially intermediary nodes, can receive smaller but stable rewards over time. In addition, this will allow each request node to download large files faster since files will come from multiple sources, rather than just a single node. For storage nodes, this extension will also reduce the risk of data loss. However, smaller files will make proof of retrievability to be more complicated because of harder verification schemes. Therefore, we may need to introduce a different challenge scheme to achieve this goal.

## References

[1] Becker, Georg (2008-07-18). "Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis" (PDF). Ruhr-Universität Bochum. p. 16. Retrieved 2013-11-20.

[2] J. Benet, D. Dalrymple, and N. Greco. Proof of replication. Technical report, Protocol Labs, July 27, 2017. https://filecoin.io/proof-of-replication.pdf. Accessed June 2018.

[3] Boneh, Dan, and Matthew Franklin. "An efficient public key traitor tracing scheme." Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 1999.

[4] Shacham, Hovav, and Brent Waters. "Compact proofs of retrievability." International Conference on the Theory and Application of Cryptology and Information Security. Springer, Berlin, Heidelberg, 2008.

[5] Benet, Juan. "Ipfs-content addressed, versioned, p2p file system." arXiv preprint arXiv:1407.3561 (2014).

[6] Labs, Protocol. Filecoin: A Decentralized Storage Network "https://filecoin.io/filecoin.pdf". 2017.

[7] QuantumMechanic. "Proof of Stake Instead of Proof of Work." BitcoinTalk, bitcointalk.org/index.php?topic=27787.0.

[8] Cohen, Bram. "Incentives build robustness in BitTorrent." Workshop on Economics of Peer-to-Peer systems. Vol. 6. 2003.