Integrity Management and Access Control of Storage Systems using Blockchain Technology

Hasan Mohammad Shahriar
Research and Development
Kona Software Lab
Dhaka, Bangladesh
h.m.shahriar@konasl.com

Muhammad Nur Yanhaona
Research and Development
Kona Software Lab
Dhaka, Bangladesh
nur.yanhaona@konasl.com

***Abstract*—Currently there is no mechanism to reliably integrate storage technologies with a blockchain network. This is a major obstacle to the adoption of blockchain technology in many real-world applications where document management is an essential element. This paper presents a novel solution for the reliable and secure integration of external document storage systems such as storage clouds and local data centers with a blockchain network. In this solution, the storage system delegates various control functions such as document access control, access fee processing, integrity insurance, and user authentication to the blockchain network through an integration gateway and focuses only on efficient storage and document delivery. This approach should facilitate greater confidence and transparency for these control functions and simplify the design of document storage systems.**

***Index Terms*—peer-to-peer computing, distributed information systems, document handling**

I. Introduction

Since its inception in 2008 [13], the blockchain technology has gained widespread attention as a transformative technology that can revolutionize many industries [1]. Blockchain based digital currencies such as Bitcoin [13] and Ether [19] are already being considered viable alternatives to existing currencies in trade and commerce for their security and ease of transfer. Blockchain smart contracts [16] [19], on the other hand, have spawned innovative applications in many business and financial sectors due to their capacity of encoding the rules of real-world interaction and ensuring their enforcement.

Central to the appeal of blockchain technology is its maintenance of a distributed ledger of transactions – called the blockchain – in a peer-to-peer network of autonomous and anonymous entities. In a blockchain network, all entities are even and none of them is trusted; still, the security and integrity of the transaction ledger can be guaranteed. This feat is achieved by a complete replication of information in all network participants where each participant validates and executes all transactions. As long as the majority of the network participants are honest, the outcome of the transactions, i.e., the state of the blockchain ledger can be trusted [14].

The blockchain technology's decentralization of trust through information and processing replication in a scalable peer-to-peer network is leading innovations and renovations in many application domains where trust and information security are key concerns. However, problem in one area in particular appears to be a major obstacle for blockchain based application adoption. This is the problem of document storage.

This research is funded by KONA Software Lab Ltd.

The blockchain technology is inherently unsuitable for storing bulky information such as files and media contents due to the networking and storage cost associated with their management. Peculiarities of blockchain ledger maintenance such as *blockchain reorg* [2] further complicates the situation by making direct integration of existing trusted storage solutions with a blockchain network difficult. Finally, the continual preservation and integrity insurance requirement for trustworthy document storage is often in conflict with the incentive for blockchain transaction ledger maintenance that rewards participants for extending the ledger of transactions only and they can join or leave the network at any time.

Nevertheless, there are some blockchain based or blockchain inspired storage technologies such as Ethereum Swarm [17], Filecoin [9], Storj [18], and IPFS [5] already available for users. These solutions break down a user's file into a series or hierarchy of data chunks then distribute the chunks to the peer-to-peer network. On a broad level, some of these storage solutions are like traditional distributed hash tables [10] [8]. Some others are like peer-to-peer file sharing services such as the popular Bittorrent [15]. These solutions apply some Bitcoin-like incentive mechanisms on top of these base technologies to motivate the network participants to retain and serve data chunks upon users' request.

The motivation for these solutions is that they protect the users from vendor locked-in and they offer an overall larger storage capacity compared to existing storage alternatives. However, blockchain based solutions have the common problem that the owner (or user) has to take the responsibility of ensuring persistence and integrity of his/her data in the blockchain by retaining document metadata and issuing periodic audits. Furthermore, despite the combined storage capacity being huge, the download bandwidth can be significantly low as the network peers may be running commodity hardware behind low-speed network connections. In addition, designing incentive mechanisms for long-term persistent of documents in a mining based blockchain network is difficult

Legacy databases of existing applications are also an obstacle for the applications' migration to the blockchain domain. Data stored in proprietary data centers are often confidential that a typical administrator may not be comfortable to put in the hands of anonymous blockchain participants. Further, when existing cloud storage providers [12] [3] have already solved the storage capacity, scalability, and cost-effectiveness problems for the clients; there is little motivation for moving data into a blockchain storage.

We believe blockchain technology should be used to introduce more transparency and better management of existing storage technologies instead of to create alternative storage solutions. In this regard, the immutable blockchain ledger can be the perfect tool to ensure integrity, to track change history, and to audit access of documents located in a storage system. On the other hand, blockchain smart contracts can be used to encode and enforce the document access constraints and to collect the storage access and usage fees.

In our scheme, location, signature, access control configuration, upload/download fees and so on metadata information about a document is stored in some blockchain smart contract, called the *document bearer contract*. A user gets access to the externally stored document by interacting with a *Storage Integration Blockchain Gateway* (subsequently referred as the *gateway*) from a blockchain client application. Any conversation with the gateway involves a series of transactions in the blockchain network for updating the document bearer contract and happens following the instructions of some secure interaction protocol. Finally, if the gateway approves the access request then it generates an access token to the external storage that the user uses to upload/download a document with the external storage directly. In case of a download, in particular, the client application verifies document authenticity by locally computing the document signature and matching it against the signature stored in the blockchain before delivering the document to the user. Figure I.1 depicts a high-level description of the system architecture of our solution.

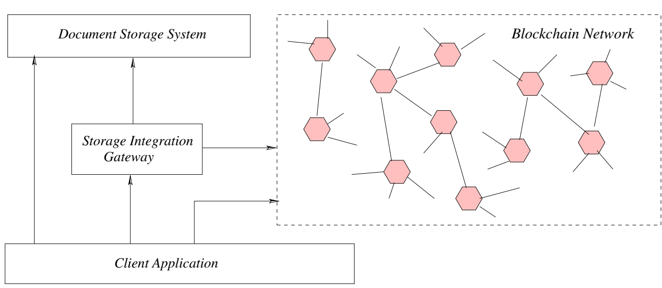

Figure I.1. External Storage System Integration Model

The arrangement of Figure I.1 can primarily be used in three different scenarios. First, a storage capacity provider can use it to introduce transparency of pricing and contractual obligation on its capacity and bandwidth usage. Second, an issuer of important documents such as a government agency can use it to define document access restrictions and to apply a paid document access strategy. Finally, a blockchain-based application service can use it to offload the cost of document storage to its users. In these scenarios, the storage provider, the document issuer, and the application service run the *storage integration gateway* respectively. These entities can be considered to be the *storage administrator* in their respective scenarios. Note that a combination of the first two scenarios is also possible where the storage provider charges the document issuer for capacity usage and the latter charges the users for document download through the same document bearer contract. These scenarios cover a broad range of use cases of storage systems in real world.

To avoid making the storage integration gateway the single point of failure regarding document access, we propose a design where the gateway database only contains information derived from the blockchain network. The sole exception is some static gateway configuration properties. This allows easy replication of the gateway and enables a gateway to resume operation after a failure without any manual update.

The underlying core innovations that make our solution work are as follows:

1) Secure and accountable document upload-download protocols with flexible blockchain based payments.
2) A generic document access control configuration paradigm using blockchain smart contracts and a mechanism to enforce the access rules in storage integration gateways.
3) Fault-tolerant design of the storage integration gateway against blockchain transaction reversal and back-end storage system failures.

This paper describes these core innovations and discusses some associated concerns. The rest of the paper is organized as follows:

A. Paper Organization

Section II discusses our problem model and its challenging ingredients; Section III describes document upload/download protocols, analyzes their characteristics, and briefs on alternative payment strategies; Section IV presents our innovation on blockchain smart contract based document access control policy configuration and its enforcement; Section V explains gateway's handling of blockchain ledger reorg [2]; Section VI discusses some related work on blockchain based/inspired document storage; Finally, Section VII concludes the paper.

II. Problem Model and Challenges

Any interaction between a storage administrator and a user of documents can only be of two types. A user either uploads or downloads a document to/from the storage system controlled by the administrator. The two participants of the interaction know each other by their blockchain identity (i.e., blockchain address) and any obligation of service between the administrator and the user is encoded in some blockchain smart contracts. In this arrangement, the parties can prove their identities to each other by simply being able to do blockchain transactions that update the relevant smart contracts. This works because any unauthenticated attempt to update a smart contract will be rejected by the blockchain network.

Given blockchain smart contract languages are Turing complete, any complex access authorization logic can also be encoded in the same smart contracts that being used for participant authentication. Then the storage integration gateway configured to do transactions on behalf of the storage administrator can perform both authentication and authorization of user access by gleaning information from the blockchain. *The challenge lies in, first, making the interaction between the user and the gateway secure and accountable for both parties; and, second, in making the access authorization logic generic to be applicable in a variety of use cases without rewriting smart contracts*.

Using the blockchain network as the veritable source of information for governing user interactions with the storage system raises the concern that the version of the blockchain a user or the gateway observes by interacting with a selective subset of network peers may not be included in the canonical longest blockchain in the long run. Blockchain ledger reorgs [2] can reverse the ledger of a peer to an alternate state at any time. Consequently, any storage access and allocation decision being made based on the old ledger state may later become disputable. *Thus the third challenge for storage integration is to make the gateway responsive to ledger reorg without violating any service agreements*.

Note that the involvement of blockchain network in document upload and download with external storages provides a natural mechanism for document integrity checking. During a document upload, a short and unique document signature can be generated from the document content and stored in the associated blockchain smart contract. During a download, the client application can recompute the signature from the downloaded content and match that with the signature found in the blockchain smart contract. If the two signatures do not match then the document has been modified or corrupted outside the guidance of the blockchain network.

Subsequent sections describes how we solve the three core challenges for storage system integration with blockchain.

III. Document Upload and Download Protocols

For the discussion of this section, we assume that both document upload and download with the external storage involves blockchain payments. Both protocols heavily use symmetric key cryptography in various steps for information and payment security. We refer the reader to [7] for an introduction to cryptography and to [6] for AES symmetric key cryptography in particular.

A. Quality Criteria for Upload/Download Protocol Design

We decide on a set of behavioral characteristics for the document upload and download protocols. These characteristics are classified into two groups based on the expectation of the storage integration gateway and that of the user from the protocols. From the gateway's perspective the following characteristics are important:

- Accountability: all actions are traceable in the blockchain.
- Independence from storage system failure: payment is collected only after all interactions with the storage system are done successfully.
- Guaranteed Payment: the user can neither fool it to do unnecessary work nor withheld the payment after the work is complete.
- Fault-tolerance: all local state information should be derivable from the blockchain ledger so that the gateway can restart easily after a failure or data corruption.
- Security from Malicious Miners: no mining nodes can snatch the payment intended for the gateway.

From the user's perspective, on the other hand, the following characteristics are desired from the protocols:

- Security: no one else can interrupt the storage session intended for the user.
- Payment Security: the user must be able to get refunds if he/she does not get the proper service.
- Confidentiality: the underlying document cannot be retrieved by others (users or miners) using the audit trail of the protocol execution in the blockchain ledger.

Propriety of both groups of behavioral characteristics is self-evident. Hence we do not elaborate on them further. During the protocol description, we discuss how different aspects of the protocols serve to achieve the mentioned characteristics.

B. Document Upload Protocol Description

The document upload protocol is composed of three main phases:

1) Token deposition by the *user*.
2) Document upload processing by the *gateway*.
3) Then, payment finalization by the *user*.

Each phase of the protocol involves multiple interaction steps among various system components. Figure III.1 presents the sequence diagram of the protocol. We now describe these steps as parts of the three protocol phases.

1) Token Deposition: The *user* generates a document key D_k by performing a hash operation on some document information (document name D_n, document uploader blockchain address DU_{addr} and document bearer contract address DC_{addr}) to identify the document. The *user* requests the *gateway* to generate a token for uploading a document with document size D_s, document hash D_h, signature of document hash DH_{sig}, DU_{addr} and D_k. The *gateway* validates the input data from the *user*. The *gateway* generates an AES symmetric key K_1 and encrypts its blockchain address G_{addr} with the generated key and produces an payment token M.

$$M = Enc(G_{addr}, K_1) \tag{1}$$

The *gateway* calculates the document upload fee according to the payment strategy and returns the generated token M and calculated upload fee to the *user*.

The *user* generates another secret key K_2, encrypts the payment token M with K_2 to produce an upload token N.

$$N = Enc(M, K_2) \tag{2}$$

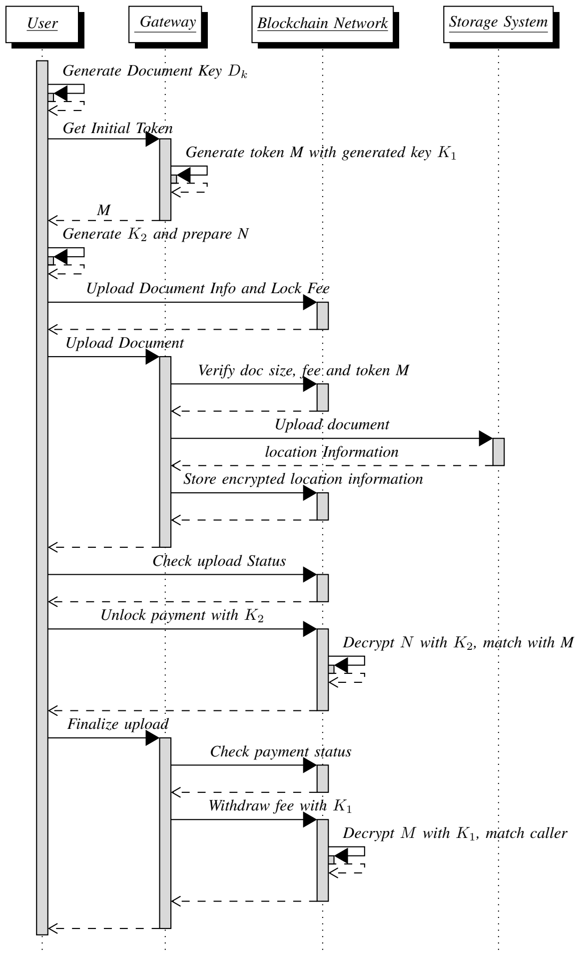

Figure III.1. Sequence diagram of the document upload protocol

The *user* performs a transaction to the blockchain with token M and N and other document meta-data. During this operation, the calculated upload fee is also deposited to the blockchain smart contract from the user account.

2) Document Upload Processing: After the earlier transaction is mined, the *user* sends the document upload request to the *gateway* with the actual document and its meta-data. The *gateway* verifies the input data and retrieves the payment token M from the blockchain and verifies that it contains the *gateway's* address G_{addr} by performing a decryption on token M with key K_1. If all verifications succeed, the *gateway* uploads the document to the storage system[1] and stores an encrypted version of the location information in the blockchain. The *user* can verify that the document upload is successful by checking the blockchain.

3) Payment Finalization: After a satisfactory verification of the outcome of upload, the *user* issues an unlock payment transaction with the *user's* secret key K_2 to the blockchain. The smart contract unlocks the payment if the decryption result of N with provided K_2 matches with stored payment token M.

$$M = Dec(N, K_2) \tag{3}$$

The user requests the *gateway* to finalize the upload. The *gateway* verifies the payment unlock status from the blockchain and issues a transaction to collect the upload payment with its secret key K_1. The smart contract decrypts payment token M with K_1 and matches the result with the caller address C_{addr}. If the address matches with decrypted result, smart contract enables the payment to the caller address.

$$C_{addr} = Dec(M, K_1) \tag{4}$$

Protocol Analysis: Since it is cryptographically hard to produce an alternative address and key pair that satisfies Equation 4, only the *gateway* can collect the payment. On the other hand, since payment is locked until someone supplies proper K_2 in the transaction evaluating Equation 3, which is only known to the *user*, the *user* ensures that the gateway cannot collect the payment until the *user* verifies that the document is successfully uploaded. In addition, since the document's location information in the external storage is recorded by the *gateway* in the blockchain in an encrypted form, none but the *gateway* can interpret this information for future reference. This simultaneously ensures information security and gateway fault-tolerance. Finally, that *user* can issue the required transactions into the designated smart contracts indirectly ensures user authentication.

Note that we did not address the possibility that the upload protocol terminates halfway in the execution for some machine or network failure. This issue can be tackled easily by associating an expiry time with the payment locking transaction. If the protocol fails before the *gateway* uploads the document in the external storage system, the user can withdraw the payment after the expiry time. If the failure happens after the document upload in the external storage, then the *gateway* can never collect the payment as the user is no longer there to unlock the payment for it. Hence, the *gateway* simply removes the document from the external storage in such cases.

C. Document Download Protocol Description

The document download protocol being illustrated in Figure III.2 consists of four phases.

1) Token deposition by the *user*.
2) Download session creation by the *gateway*.
3) Payment approval by the *user*.
4) Finally, document download by the *user*.

The phases are described in more detail below.

1) Token Deposition: The *user* requests the *gateway* to generate a document download payment token with the document key D_k, document hash D_h, *user's* blockchain address U_{addr}, and signature of document hash DH_{sig}. The *gateway* verifies the input data, generates a payment secret key K_a and encrypts its blockchain address G_{addr} with the secret key and produces a payment token T_p.

$$T_p = Enc(G_{addr}, K_a) \tag{5}$$

[1] If the storage system allows gateway defined transferable upload sessions for users then the user can do the document upload instead of the gateway.

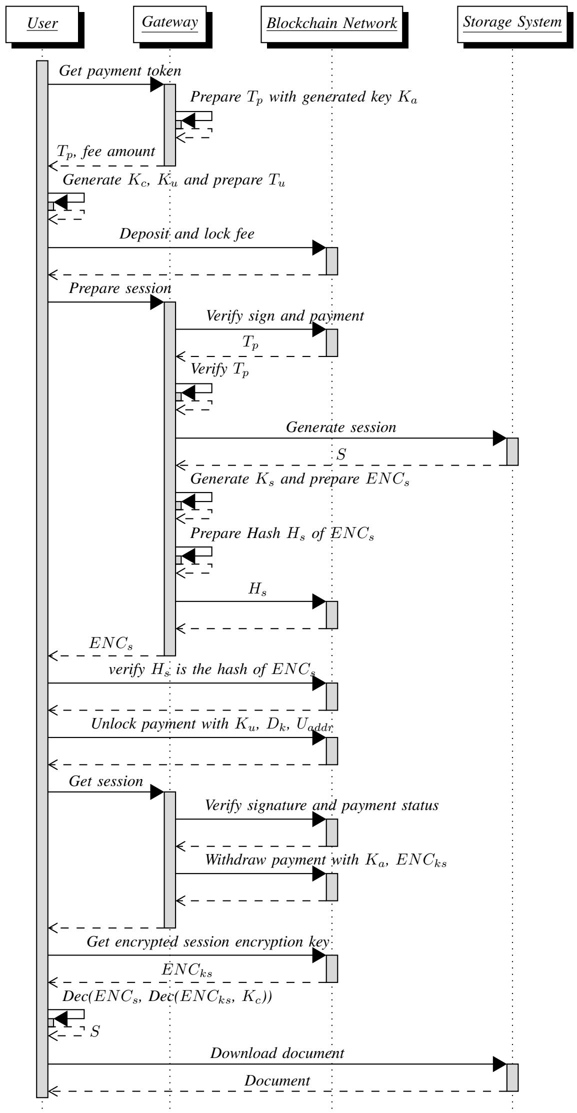

Figure III.2. Sequence diagram of the document download protocol

The *gateway* also calculates the fee according to the payment strategy to download the document and returns the payment token T_p with the calculated download fee to the *user*. After receiving the payment token, the *user* generates two symmetric keys K_c and K_u. K_c is the common conversation secret shared only with the *gateway*. The *user* generates an payment unlock token T_u by encrypting T_p with the key K_u.

$$T_u = Enc(T_p, K_u) \tag{6}$$

The *user* then issues a download payment transaction to the blockchain with two tokens T_p and T_u, and some document related information that deposits the download fee in the blockchain smart contract.

2) Download Session Creation: After the download payment transaction is mined, the *user* issues a prepare download session request to the *gateway*. During this request, the *user* sends document bearer contract address DC_{addr}, signed document key DK_{sig}, U_{addr}, D_k, D_h, and K_c. The *gateway* validates the input data, retrieves the payment token T_p from the blockchain and verifies by decrypting it with K_a and matching the result with its own blockchain address G_{addr}.

$$G_{addr} = Dec(T_p, K_a) \tag{7}$$

After successful address verification, the *gateway* retrieves the external storage information for the document and requests the *storage system* for a session S to download the document. The *gateway* then generates a secret key K_s, encrypts S with K_s and produces an encrypted session ENC_s. It also encrypts the document key with the common secret K_c and produces an encrypted document key ENC_{dk}.

$$ENC_s = Enc(S, K_s) \tag{8}$$

$$ENC_{dk} = Enc(D_k, K_c) \tag{9}$$

The *gateway* then prepares a hash, H_s, of ENC_s and issues a transaction to the blockchain with H_s and ENC_{dk}. The *gateway* locally stores all input data and returns ENC_s to the *user*.

3) Payment Approval: The *user* verifies that the hash of ENC_s is stored in the blockchain by the *gateway* then issues a transaction to unlock the payment with K_u, D_k and U_{addr}. The smart contract unlocks the payment by decrypting the unlock token T_u with the secret key K_u provided by the *user* and matching the result with the payment token T_p.

$$T_p = Dec(T_u, K_u) \tag{10}$$

The *user* then issues a request to the *gateway* to get the download session encryption key with D_k, DK_{sig} and U_{addr}. The *gateway* verifies the input data and encrypts the session secret K_s with common conversation secret K_c and produces an encrypted session encryption key ENC_{ks}.

$$ENC_{ks} = Enc(K_s, K_c) \tag{11}$$

The *gateway* issues a blockchain transaction to withdraw the payment with the K_a and ENC_{ks}. The blockchain smart contract enables the download payment fee to the caller address if the decrypting result of payment token T_p with K_a matches with the caller address. It also stores ENC_{ks} with the download document information in blockchain.

4) Document Download: The *user* then collects the encrypted session encryption key ENC_{ks} from the blockchain. The *user* first decrypts the encrypted session encryption key ENC_{ks} with the common secret K_c and retrieves the session secret key K_s, then decrypts the encrypted session ENC_s with K_s to retrieve the original session S.

$$K_s = Dec(Enc_{ks}, K_c) \tag{12}$$

$$S = Dec(Enc_s, K_s) \tag{13}$$

Finally, the *user* downloads the document directly from the storage system using the session S.

Protocol Analysis: Payment security for both the *user* and the *gateway* is ensured as it was in the upload protocol using a pair of encrypted tokens and proper sequencing of the payment related blockchain transactions. Likewise, all steps of the download protocol are also reflected on the blockchain smart contract. This ensures gateway accountability. Further, the *gateway* again collects the payment only after completing its interaction with the *storage system* to protect itself from any blame related to latter's failure. Finally, information related to the storage session is recorded in the blockchain in encrypted form while the key K_c for decrypting it is shared securely between the *user* and the *gateway*. Hence none but these two entities can use the session to get access to the document.

One aspect of the download protocol requires particular attention. This is related to unlocking of the download fee by the *user* before he/she can verify that the storage session related information supplied by the *gateway* is accurate. Since the *user* can be malicious, payment must be unlocked for the *gateway* before the *user* can decrypt the external storage session related information. However, this ordering opens the possibility that the *user* ends up paying for an invalid storage session or dispute its validity when it is actually valid. Therefore, a mechanism is required for automatically resolve any dispute related to external storage session in the blockchain smart contract. This is facilitated by the series of encryption related to the conversation secret K_c and the storage session S.

To submit a claim about an invalid session, the *user* sends K_c and ENC_s in the blockchain smart contract. That the user has supplied the valid conversation secret is verified by performing the inverse of Equation 9 and matching the answer with the document key D_k. Since it is the *gateway* who did the original transaction, the *user* also establishes that the provided K_c value was known to the *gateway*. Now the blockchain smart contract itself can compute Equation 12 and 13 to retrieve the session S. It then computes a hash of S and checks if that matches H_s which being stored in the blockchain by the *gateway*. If both hashes matches then the *user's* session invalidity claim is serious. Then the *gateway* maliciousness and external storage malfunctioning should be investigated. Otherwise, the *user's* claim is ignored.

IV. Document Access Permission Control

A. Access Control Policy Configuration

Since a user requesting access to the storage system for upload/download identifies him/herself with his/her blockchain address, for a solely blockchain-based access authorization, the storage integration gateway should be able to somehow relate a user's blockchain address to the properties of document bearer smart contract. Ideally, the required relation should be defined in the said bearer contract. To provide a generic solution for expressing relationships between contracts and users that covers most scenarios, we treat each document bearer contract as a part of one or more rooted contract relationship hierarchies. Then we provide a path expression language to indicate what contracts reachable from the document bearer contract through different paths contain information about users having some form of access to the underlying document or the storage allocation dedicated for it.

To understand this concept, consider the scenario where the storage capacity is used to support a blockchain application for crowd-funded real-estate asset developments where investors receive ownership rights on parts of the assets. The asset developer receives the profit for an asset development. However, the actual development work is conducted through a series of activities assigned to sub-contractors. Further, some regulatory agency is tagged with each asset development project by the local branch of the ministry of land for overseeing purposes. Finally, the application allows an investor to trade his/her share on the asset for profit. Then a contract relationship hierarchy for this description can be as illustrated in Figure IV.1. In this description, the root of the relationship is the *Project* contract.

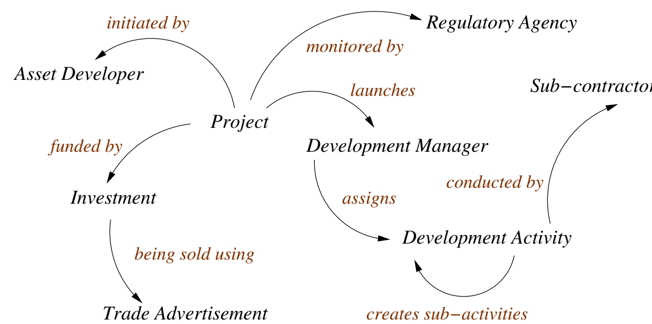

Figure IV.1. An example smart contract relationship hierarchy

Now if the blockchain application requires that an asset developer can download "business profile" documents of only those sub-contractors who contributed to his/her asset development projects, this requirement can be expressed in the sub-contractors' smart contracts as a path connecting properties of intermediate contracts and ultimately leading to the asset-developer. The advantage of using path expressions instead of using blockchain addresses of corresponding entities directly is that if their relationship is broken due to some property update in the intermediate contracts, the document access permission is automatically revoked. Likewise, new users are granted access to documents automatically as new relationships are established in the blockchain.

To make this scheme work in the current Ethereum smart contract specification, we only need that each smart contract belongs to some predefined contract template and bears with it a template type identifier and a method to list all its properties. If there was a mechanism to search the property names of an arbitrary smart contract deployed in the blockchain, the template typing would not be required. Anyway, we believe this is not a difficult requirement to satisfy.

Figure IV.2 describes the grammar for permission policy configuration for a document bearer smart contract. The grammar requires that access permissions are specified per document type and individually for each approved user category. There are three components of each permission expression that

⟨permission⟩	⊨	⟨docperms⟩ | ⊥
⟨docperms⟩	⊨	⟨docperm⟩ | ⟨docperm⟩ ; ⟨docperms⟩
⟨docperm⟩	⊨	⟨doctype⟩ ⟨userperms⟩
⟨userperms⟩	⊨	⟨userperm⟩ | ⟨userperm⟩ AND ⟨userperms⟩
⟨userperm⟩	⊨	[⟨permbits⟩] ⟨permexpr⟩
⟨permexpr⟩	⊨	⟨localexpr⟩ | ⟨remoteexpr⟩
⟨doctype⟩	⊨	⟨attrname⟩ : ⟨attrmult⟩ –
⟨permbits⟩	⊨	r- | -w | rw
⟨localexpr⟩	⊨	(⟨attrname⟩ : ⟨attrmult⟩)
⟨remoteexpr⟩	⊨	⟨acclink⟩ / ⟨srclink⟩ / ⟨overlap⟩
⟨acclink⟩	⊨	accessor (⟨attrmult⟩) [⟨pathdirection⟩] =⟨path⟩ = root
⟨srclink⟩	⊨	this = ⟨path⟩ = root
⟨overlap⟩	⊨	none | substr
⟨pathdirection⟩	⊨	F | R
⟨path⟩	⊨	⟨edge⟩ | ⟨edge⟩ – ⟨path⟩
⟨edge⟩	⊨	⟨linkerprop⟩ : ⟨contracttype⟩ ⟨occurrences⟩
⟨attrname⟩	⊨	⟨string⟩
⟨attrmult⟩	⊨	single | array
⟨linkerprop⟩	⊨	⟨string⟩
⟨contracttype⟩	⊨	⟨string⟩
⟨occurrences⟩	⊨	⊥ | [*]

Figure IV.2. Permission Expression Grammar

grants a specific kind of users an specific kind of access to a specific document of a specific type of smart contract:

1) Source Linkage: a path description that tells how the root of the contract relationship hierarchy can be reached from the document bearer contract.
2) Accessor Linkage: another path description that tells how the root of the contract relationship hierarchy can be reached from the contract holding the user address.
3) Path Intersection Requirement: a specification that tells to what extent the source and accessor linkages should overlap to grant the user access to the document.

For example, assume in the contract relationship hierarchy of Figure IV.1, the *investor* of any *Investment* contract, the *assigner* of an ancestor *DevActivity* contract, and any of the *representatives* of the *RegulatoryAgency* contract can view the *license* document of the *Subcontractor* contract corresponding to the *assignee* of a *DevActivity* of the underlying project. Meanwhile, only the *owner* of the Subcontractor contract can both read and replace (i.e., write) the *license* document in the external storage. Then the permission configuration for the *license* document should be as described in Listing 1:

```
license:single —
[r−]: accessor(single)[F] = investor:Investment — project:Project = root /
  this = assignee:DevActivity — parent:DevActivity[∗]
  — manager:ActivityManager — project:Project = root / none
AND [r−] accessor(array)[R] = representatives:RegulatoryAgency
  — regulator:Project = root /
  this = assignee:DevActivity — parent:DevActivity[∗]
  — manager:ActivityManager — project:Project = root / none
AND [r−]: accessor(single)[F] = assigner:DevActivity—parent:DevActivity[∗]
  — manager:ActivityManager — project:Project = root /
  this = assignee:DevActivity — parent:DevActivity[∗]
  — manager:ActivityManager — project:Project = root / substr
AND [rw]: (owner:single)
```

Listing 1. Example access permission configuration

Our strategy for specifying the access permissions using contract relationship path expressions is a middle ground between access control list (ACL) and role based access control (RBAC) that are typically being used in file systems [4]. Like RBAC, users gain access to resources because they held specific roles due to their association with specific smart contracts. Unlike generic roles in RBAC, however, roles in our case are indirectly bound to specific resources through some relationship graphs.

B. Access Control Policy Enforcement

A storage integration gateway does not need to understand specific smart contract relationship hierarchies to enforce access control rules specified in the permission expressions of various smart contracts. It only requires that the blockchain network notifies it whenever a smart contract is deployed and contracts emit properly formatted events[2] in the blockchain audit log in response to blockchain transactions that may affect any source or accessor linkages.[3]

The gateway monitors these events and derives a document permission database by combining event data and gateway's path expression related query results. This database is like a dynamic ACL database that is automatically being updated in response to new blockchain events. Figure IV.3 depicts a high level entity relationship diagram of the gateway database. As new smart contracts are being deployed in the blockchain

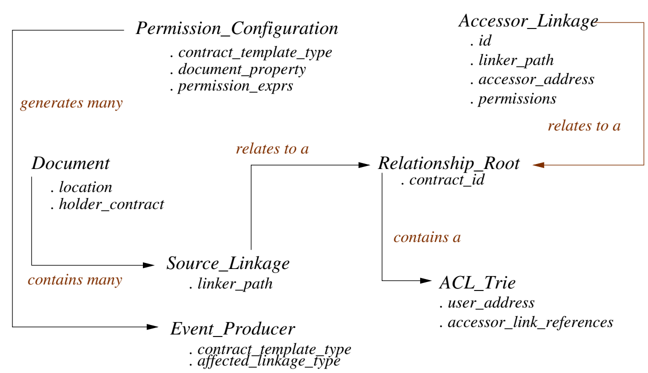

Figure IV.3. Entity Relationship Diagram of the Access Permission Database

network, the gateway updates contracts' template related information in local database to determine if a contract of this type can affect any document access permission through some source and accessor linkages. The gateway also determines if this type of contract can be a root of any contract relationship hierarchy. For each contract that is a root of a relationship hierarchy, the gateway creates an ACL holding Merkle tree

[2]An event should contain the addresses of the two contracts whose direct association has been affected by the blockchain transaction, their template type names, and the nature of the update.

[3]If we consider a smart contract relationship hierarchy as a directed graph, all relationships may not be traceable from the leaf to the root of the hierarchy. Some relationships may go backward from the root to the leave. Regardless, the events on the audit log should be formatted to support a uniform traversal strategy from the leave to the root.

[11] that contains the user addresses that are the endpoints of various accessor linkages originated from the root.

If a document is uploaded or an event is published about a change in any property that contributes an edge to the document's paths to various relationship roots, new source linkages are being computed and stored in the database. At the same time, invalidated source linkages are being eliminated. Similarly, if any smart contract property that contributes an edge to users' accessor path to various relationship roots, valid accessor linkages are being recomputed. In addition, the ACL tree entries of the affected users are also being updated.

When a user requests the gateway to undertake an upload/download protocol with the user, the gateway receives the underlying document bearing smart contract's address, the attribute of the smart contract the document refers to, and the user address. Since in case of an upload, the user must be associated with the document bearing smart contract directly, the gateway merely checks the smart contract template description to determine what query to issue in the smart contract to search for the user. If the requesting user is found registered as a valid uploader, the upload protocol initiates. Once upload is done, blockchain audit log is traversed to determine new source linkages for that document to existing contract relationship roots. These linkages are then inserted in the gateway database. If the new document replaces some document uploaded earlier then only the location related metadata need to be updated in the gateway database to point to the new document.

If the user requests for a document download session, the gateway first identifies the ACL trees correspond to smart contract relationship roots reachable by traversing the source linkages of the document. Then it checks if the user's address is in any of those trees. Then it retrieves all accessor linkages ending at the user address in various trees. Finally, related document source and accessor linkages are compared to decide if they can be combined satisfying any read permission configuration for the document. If a combination attempt succeeds then the user is granted access. If the authorization process fails in any step then the user's request is denied.

V. Handling Blockchain Ledger Reorganization

Without loss of generality, for the discussion of this section, we assume that the storage integration gateway connects with a single peer in the blockchain network.

When a blockchain ledger reorganization (blockchain reorg in short) replaces the peer's ledger the gateway is connected with, the protocol for blockchain updates ensures that the new ledger the peer has accepted is longer than the old ledger. Now, the new ledger may have all or some the same transactions affecting access control decisions and upload/download protocol executions – albeit may be in different blocks; or it may have none of those transactions. So the gateway first need to assess to what extent it has been affected by a blockchain reorg then takes necessary corrective actions.

To facilitate detection of the extent of a blockchain reorg, the gateway maintains a sequence of block-hashes starting from the genesis block and ending at what the gateway believes to be the last mined block in the canonical longest chain[4]. Whenever it receives a notification of a new block being added in the existing chain of its connected peer, the gateway extends the block-hash sequence with the new block's hash. The gateway easily identifies that a blockchain reorg has happened in the peer when it detects that the new block being reported from the peer has a parent hash different from the head of its own block-hash sequence. In that case, the gateway continues to request blocks from the peer in the reverse order until a hash matching an entry in its block-hash sequence is found. Then the top non-matching subsequence of block-hashes represents the extent of the blockchain reorg. The gateway then removes this subsequence from its block-hash sequence and starts reorg processing. The gateway's after-reorg corrective actions for access control database updates differ from that for its upload/download protocol execution steps restoration.

A. Correcting the Access Control Database

The gateway has a simple strategy for updating its access control database after a blockchain reorg. It has each database record tagged with the hash of the block that causes the record's creation or its latest update, and it maintains older versions of its mutable records in deleted status in the database. Then when the block-hash subsequence for reversed blocks has been received, it deletes all records (active/deleted) that have a matching tag in the subsequence. Then it makes the latest remaining deleted records of mutable records active.

The gateway database now reflects an earlier state of the current blockchain ledger of its connected peer. As if the gateway was sleeping and unaware of any subsequent block mining. The gateway then continuously asks the peer for newer blocks and keeps processing the events from their audit log. When there is no more blocks to process, the gateway database is consistent with the current blockchain ledger of the peer.

B. Restoring Protocol Execution Steps

In Ethereum-like blockchains, out of order transactions from a single blockchain address cannot exist in any blockchain ledger. However, since both document upload and download protocols involve blockchain transactions from both the gateway and the user, it is possible that after a blockchain reorg, the new ledger of the connected peer has some of the user's transactions but none of the gateway's. If this happens then the blockchain information regarding a protocol execution will become inconsistent, consequently, disputable.

To avoid such inconsistency, a document bearer smart contract maintains a simple linear state machine for each ongoing protocol session with a user. The state machine ensures that at the i^{th} state of the contract, only the transaction for the $i+1$ protocol step is accepted in the blockchain. With this update, the blockchain reorg can only rollback a protocol transactions sequentially from the last step.

The gateway logs each step of a protocol execution in its local database tagged with the hash of the block that included

[4]Note that a block's hash uniquely identifies the block in the blockchain.

the corresponding transaction. By matching the block-hash subsequence for reversed blocks with the hashes stored in the database, it determines what protocol executions have been affected by the reorg. Now, instead of deleting the the database entries tagged with invalidated block-hashes as in the case of the access control database update, the gateway first tries to update their tags to account for steps whose transactions have been moved to different blocks due to the reorg[5]. The entries that could not be updated even after the gateway has received the most recent block from its peer are then marked as invalid.

Note that the upload and download protocols both have the property that except for the first transaction that the user issues to transfer coins from his/her account to lock them into the document bearer smart contract, all other transactions are cryptographic key based. So anyone else having the keys can re-execute them on behalf of the user. Furthermore, the user's cryptographic keys are all revealed in the blockchain ledgers after the completion of a protocol execution. Hence, the gateway retrieves the user's keys from the blockchain ledger during the protocol execution and stores them with associated step log entries in its database. Then if it is found that a protocol execution trace has not being completely erased due to step rollback, the gateway simply issues the transactions for the missing steps to restore the blockchain trace and collects any associated payment. That the gateway could restore the protocol execution trace in the blockchain is a sufficient proof that the user has participated in the protocol at an earlier time.

If all steps of a protocol execution have been rolled back, this is an irrecoverable situation for the gateway as the gateway can never collect its payment nor can it prove that it has already rendered a service to the user. in case of an upload, the current solution schedules the document to be removed from the storage system after some time. Instant removal is not done because another blockchain reorg may give the gateway a better chance for protocol execution restoration. Nothing could be done in case of a download. Improving the protocols for tackling this scenario is a subject of future research.

VI. Related Work

To the best of our knowledge, alternative solutions for integrating external storage systems with a blockchain network do not exist. So we describe the well-known blockchain based storage solutions for the sake of completion.

IPFS [5] is a popular blockchain inspired distributed storage solution. It is basically a distributed hash table (DHT) [10]. The content of each file in IPFS is broken into fixed size blocks and distributed in the network. The hash of a block's content uniquely identify the block in the network. Participant nodes in a IPFS network cache each others data using a Bittorrent [15] inspired protocol called BitSwap. In BitSwap, each node has a balance that represents the sum of the ratios of the number of blocks from other nodes it caches compared to the number of its own blocks those other nodes cache. Nodes with large negative balances gradually become isolated in the network by their peers. This encourages a good caching behavior.

Like IPFS, Swarm [17] is also a DHT where files are divided into chunks and distributed among the participant nodes in the network. Unlike IPFS, nodes in Swarm receive cryptocurrency payments for serving those chunks to the requester. In addition, a node can make a promise for long term storage of a chunk by issuing a promissory note in the form of a blockchain smart contract. If the node fails to meet its promise, the original owner of the chunk can submit the promissory note as an evidence of misconduct and receive a compensation payment. The integration of payment and penalty in Swarm makes it less likely than in IPFS that nodes will drop chunks.

Like Swarm, Storj [18] storage network stores files by dividing its content as fixed sized shards and distributing those shards among the network peers. A network peer, called a farmer, gains Storj coins by serving those shards on user request. Unlike Swarm, there is no built-in provision for contractual agreement between the file owner and the farmer storing a shard. Instead, the owner does periodic audits of the existence of shards using some file metadata. For safety, the metadata for conducting an audit can be stored in some secondary blockchain. Another major difference from Swarm is that all Storj chunks are encrypted by the owner before he/she send them to the farmers.

Filecoin [9] seamlessly integrates data storage concerns with blockchain network maintenance by making storing of file content a prerequisite for block mining using a scheme called *Proof of Retrievability*. Here again a file is divided into fixed sized pieces and distributed among the network peers. In addition, a blockchain ledger is maintained by the peers that records all transactions regarding store and access requests issued by clients. There is a deterministic algorithm for choosing a small subset of existing pieces from different files whose data is used as the input for the next block mining challenge. Therefore, if a peer stores more file pieces, its chance of success in block mining increases. Consequently, peers are inclined to hoard pieces and serving them.

All these solutions make the file owner responsible for insuring the confidentiality, integrity, and long term availability of his/her file content. In addition, since pieces of a file coming from different parts of the network need to be stitched together before serving, file download latencies can be significant and unpredictable. Finally, since the loss of a single piece corrupts the entire file, all pieces must be stored with the same level of redundancy. That may increase cost of storage significantly.

VII. Conclusion

This paper serves as a comprehensive description of our solution for securely integrating document storage systems such as storage clouds and data centers with a blockchain network. Access to those storage systems is governed by policies regarding payments and user permissions specified in blockchain smart contracts. The solution positions a gateway system in between the storage and the blockchain network that

[5]Updating is possible because the document bearer contract emits an event in the blockchain audit log during each step execution.

enforces these policies. The gateway is designed to be easily replicable and tolerant to internal failures.

The solution's philosophy is that regarding the management of bulky and, in particular, static information such as documents and images, the blockchain technology should take the responsibility for access control, payment processing, document integrity insurance, and auditing while a storage technology should hold the coveted information and provide access to them. This partitioning of responsibility provides the flexibility of using a mature storage technology in blockchain powered applications. This provides an added security and accountability for document processing that currently only the blockchain technology can offer. Our solution is quite different from the blockchain based storage alternatives that store bulky information directly within the blockchain network using complex mining incentive mechanism and still burdens the user with information availability and integrity insurance.

There are three aspects of our solution: first, cryptographically secure and accountable protocols for handling information upload and download with the external storage; second, a generic access permission control strategy for launching those protocols; finally, a mechanism to deal with reversal of blockchain transactions related to the protocols and access control. The paper described all three of them.

References

[1] Break through with blockchain. https://www2.deloitte.com/us/en/pages/financial-services/articles/blockchain-series-deloitte-center-for-financial-services.html. Accessed: 2019-05-06.

[2] Chain reorganization. https://en.bitcoin.it/wiki/Chain_Reorganization. Accessed: 2019-02-12.

[3] Google cloud storage: Features and benefits. https://cloud.google.com/storage/features/. Accessed: 2019-05-15.

[4] John Barkley. Comparing simple role based access control models and access control lists. In *Proceedings of the Second ACM Workshop on Role-based Access Control*, RBAC '97, pages 127–132, New York, NY, USA, 1997. ACM.

[5] Juan Benet. Ipfs - content addressed, versioned, p2p file system, 07 2014.

[6] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, 1999.

[7] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979.

[8] Michael J. Freedman and David Maziéres. Sloppy hashing and self-organizing clusters. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, pages 45–55, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[9] Protocol Labs. Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf, 2017.

[10] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[11] R. C. Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122, April 1980.

[12] James Murty. *Programming Amazon Web Services*. O'Reilly, first edition, 2008.

[13] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[14] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 643–673, Cham, 2017. Springer International Publishing.

[15] Johan Pouwelse, PawełGarbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Proceedings of the 4th International Conference on Peer-to-Peer Systems*, IPTPS'05, pages 205–216, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[17] viktor trón, aron fischer, dániel a. nagy, zsolt felföldi, and nick johnson. swap, swear and swindle incentive system for swarm. https://swarm-gateways.net/bzz:/theswarm.eth/ethersphere/orange-papers/1, May 2016.

[18] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network. https://storj.io/storj2014.pdf, 2014.

[19] D. Wood. Ethereum: a secure decentralised generalised transaction ledger. 2014.