Transparent, Distributed, and Replicated Dynamic Provable Data Possession

Mohammad Etemad and Alptekin Küpçü
Koç University, İstanbul, Turkey
{metemad, akupcu}@ku.edu.tr

Abstract

With the growing trend toward using outsourced storage, the problem of efficiently checking and proving data integrity needs more consideration. Starting with PDP and POR schemes in 2007, many cryptography and security researchers have addressed the problem. After the first solutions for static data, dynamic versions were developed (e.g., DPDP). Researchers also considered distributed versions of such schemes. Alas, in all such distributed schemes, the client needs to be aware of the structure of the cloud, and possibly pre-process the file accordingly, even though the security guarantees in the real world are not improved.

We propose a distributed and replicated DPDP which is transparent from the client's viewpoint. It allows for real scenarios where the cloud storage provider (CSP) may hide its internal structure from the client, flexibly manage its resources, while still providing provable service to the client. The CSP decides on how many and which servers will store the data. Since the load is distributed on multiple servers, we observe one-to-two orders of magnitude better performance in our tests, while availability and reliability are also improved via replication. In addition, we use persistent rank-based authenticated skip lists to create centralized and distributed variants of a dynamic version control system with optimal complexity.

1 Introduction

In recent years, cloud storage systems have gained considerable attention from both academia and industry, due to the services it can provide at lower costs. As a result, IT outsourcing has grown by 79% [5]. In the case of outsourcing storage, the client wants to upload her data to a server, and wants to rest assured that her data remains intact. She may trust the server in terms of availability, but does not necessarily trust him to keep her data intact. Indeed, the server may try to hide data loss or corruption due to hardware or software failures. When the data is large, it is not acceptable to require the client to retrieve the whole file in order to validate it, since this requires high bandwidth and time complexity [2]. This will be even more problematic if the client uses resource-constrained devices, or performs this check frequently [19].

Ateniese *et al.* [2] proposed the concept of *provable data possession* (PDP), which provides probabilistic guarantees of possession of the outsourced file. In PDP, the file is divided into blocks before being stored on the server, and later, the client may challenge a randomly-selected subset of the blocks. The server's response to this challenge is then verified by the client to check the integrity of the file. Juels and Kaliski [23] developed a similar model named *proof of retrievability* (POR). The route is followed by others [34, 37, 33, 16, 4], alas, only for static files.

Later, dynamic cloud storage protocols were developed by Erway *et al.* [19] and Ateniese *et al.* [3], and later variants followed [36]. The DPDP scheme [19] uses rank-based authenticated skip list, which supports insertion, modification, and deletion of blocks in $O(\log n)$ time, where n is the number of blocks.

All these schemes deal with integrity checks, but if the data is lost, it can only be detected, not recovered. The instant solution to this problem is to store multiple copies of the file, and use other copies if one is corrupted. Many such solutions exist for both static and dynamic scenarios [14, 10, 5, 7, 40, 41] but these schemes require the client to perform pre-computation that is on the order of the number of servers/replicas (e.g., generate multiple encoded copies of the file), and the CSP architecture is not transparent from the point of view of the client.

This imposes an unnecessary burden on the client, decreasing her efficiency, while she has no way to check whether the CSP keeps storing exactly the agreed-upon number of replicas, unless the client interacts with each server (storing a replica) one-by-one. Even in that case, presumably the inter-server communication is much faster than the client-server communication, thus a single server in the background may be storing the data and providing proofs to multiple servers interacting with the client. Moreover, if the client takes into account the internal architecture of the CSP in the pre-computation, then the CSP cannot even improve his architecture without notifying the client (which leads to re-computations). Normally, the CSP wants to change his structure and adapt it with the world technical progress (e.g., Amazon S3 is said to store three replicas for each file [38], which may be increased or decreased with technological advancements).

In this paper, we propose a *transparent*, distributed, and replicated dynamic PDP (DR-DPDP), based on the DPDP scheme [19] (or its variants following a similar structure). The CSP's architecture is completely transparent to the client, and hence the client performs in the same way as in DPDP.

Our DR-DPDP scheme *does not decrease the guarantee on detection probability*, and hence incurs no harm to the client, while helping her get rid of pre-computation imposed solely by the architecture, and later checking data integrity toward a specific architecture. We improve the client's efficiency, and achieve *better scalability, availability and reliability* at the CSP. The CSP can flexibly manage its resources, perform its own choice of load balancing and replication schemes in the background, while still providing *provable* storage for the client. This makes DR-DPDP much easier to deploy on real systems.

We also present a provable Version Control System (VCS), achieving better, indeed optimal, complexity $O(1 + \log n)$. We further combine our DR-DPDP scheme with our VCS to obtain a distributed VCS (DVCS) scheme with the same complexity.

Contributions. The main contributions of this paper are as follows:

- We propose the first *transparent*, distributed, and replicated provable dynamic cloud storage system.
    - Our system incurs no cost over the single-server case; it actually *improves the performance* due to parallelism. Indeed, for a system with 100 servers and 10 replicas, our system performs *10 times faster* updates and proofs than a single-server storage.
    - Asymptotic complexity of our system does *not* depend on the number of replicas, servers, or partitions.
    - The CSP's architecture is completely transparent to the client, and can be *changed on-the-fly* with the sole decision of the CSP. Therefore, our clients and servers are more efficient due to the transparency of the server's architecture.
- We present a (distributed) version control system with optimal complexity.

– We map VCS operations to underlying cloud storage operations in a provable manner for the first time, and show that, in many cases, the complexity of the operation is independent of the number of versions.
– We consider the multi-client scenario for provable VCS for the first time.

1.1 Related Work

Proof of Storage. A trivial way to perform integrity check is via message authentication codes or hash functions. The client hashes her file before uploading, and stores the hash value. Later, the client retrieves the whole file, recomputes its hash, and checks if it matches the stored hash value [5]. This is not efficient since each verification requires the whole file to be transmitted. The client can divide the file into blocks, hash each block separately, and challenge a random subset of blocks. Again, all challenged blocks should be transmitted for verification [2].

An asymptotically-efficient hash-based scheme is proposed by Deswarte *et al.* [15] with $O(1)$ complexity for client storage and communication. But, since the operation at the server is exponentiation with the entire file in each verification, the system will be extremely slow when used with large files.

Ateniese *et al.* [2] proposed a PDP scheme with which these efficiency problems have been solved. In PDP, first the client divides the file F into n blocks ($F = f_1|f_2|...|f_n$), then computes a *tag* for each block, and finally transfers the file along with the tags to the server, deleting its local copy. Later, she sends a *challenge*, which is a pseudorandom function key used to generate random block indices and coefficients, to the server. Upon receipt, the server constructs a *proof* using the tags and blocks stored, and the random block indices and coefficients, and sends the proof back to the client for verification.

Juels and Kaliski [23] proposed a POR scheme where the main difference was that the client uses erasure codes to encode her file before uploading. This enables resilience against data losses at the server side: the client may reconstruct her data even if the server corrupts (deletes or modifies) a portion of it.

The PDP and POR, as well as their later variants and generalizations [34, 16, 4] support only static files. The first dynamic schemes were proposed simultaneously by Ateniese *et al.* [3] and Erway *et al.* [19]. Ateniese *et al.* [3] proposed the Scalable PDP, where the client pre-computes responses for pre-decided challenges and stores them on the server encrypted. This means that the number of updates and challenges a client can perform is limited and fixed, and is determined by the client during pre-computation. They achieve optimal asymptotic complexity $O(1)$ in all complexity criteria (server computation, client computation, communication, server storage, client storage), yet each update requires rebuilding all the remaining challenges. (Note that the number of challenges is also considered a constant).

Erway *et al.* [19] proposed a Dynamic PDP (DPDP) scheme in the standard model that supports provable unlimited updates (modify, delete, and insert) with $O(\log n)$ complexity. The scheme is based on *rank-based authenticated skip list*, in which, only the relative indexes of blocks are used, so it can efficiently support dynamism. The proof for a block is computed using values in the search path from that block up to the root of the skip list. Since a skip list is a tree-like structure that has probabilistic balancing guarantees, the proofs will have $O(\log n)$ complexity with high probability.

All these schemes deal with the *integrity* of the outsourced data, but the *availability* and *reliability* are important as well. One method to support availability and reliability is to store several copies of the file, resulting in better availability and efficiency. A first effort to use multi-copy paradigm was the MR-PDP (Multiple-Replica PDP) scheme proposed by Curtmola *et al.* [14] that extends the (single-copy) PDP. MR-PDP enables a client to store t replicas of a file in a storage system, and to verify through a challenge-response protocol that each unique replica is accessible. HAIL [10], on the other hand, distributes POR to multiple servers, trying to balance their load.

Barsoum *et al.* [5] proposed a multi-copy PDP scheme for *static* files, where the client generates t distinct copies of the file by encrypting the file under t different keys. Later, she separately challenges each copy using a PDP scheme to ensure that the cloud is possessing all t copies. Hence, the scheme is simply applying a PDP scheme to t different files, but the efficient scheme uses Homomorphic Linear Authenticators [4] that enables the server to aggregate challenged tags to construct an efficient proof.

The client first generates t copies of the file $\{F_1, F_2, ..., F_t\}$, then generates an aggregated tag for the blocks at the same indices in each copy, then sends all to the server, deleting its local copy. Later, the client issues a single random PDP challenge as discussed above. Upon receipt, the server computes an aggregated proof, which shows that he is still correctly possessing all t copies, and sends it back to the client for verification. Here, *multi-copy does not necessarily mean multi-server*: a single server may store all copies, making the system vulnerable to failures.

Barsoum *et al.* later proposed two multi-copy DPDP schemes, based on tree and map structures [7, 6]. The tree-based scheme uses the Merkle hash tree [26], which is a binary tree used to authenticate the values of the data blocks via collision-resistant hash functions. In this scheme, each copy of the file is put into a Merkle hash tree, and their roots are used to construct another Merkle hash tree (the *directory*) whose root is the metadata stored at client, similar to the file system proposal of DPDP [19]. Again, the challenge-response mechanism is not transparent to the client; she must know this new structure is in place.

Zhu *et al.* [40, 41] proposed a Cooperative PDP where the client generates the tags of the blocks and then sends them securely to the *organizer*. The organizer is one of the servers who is responsible for communication with the client, and determines on which server each part of file will be stored. Later, when the client challenges the organizer, he gathers together responses from multiple servers and sends a single final response back to the client. Since the responses are homomorphic, the organizer can combine them efficiently. A disadvantage is that the distributed cloud architecture is not transparent to the client, since the tags depend on which server is storing a particular block.

In our DR-DPDP scheme, the client is exactly a DPDP client, and all servers storing data are DPDP servers. The decision about the distribution of partitions over servers, number of servers, replication, etc. are all up to the CSP. Most importantly, the whole process is transparent to the client, and she is still guaranteed that there is at least one intact copy.

It is important to note that *the CSP may store all the data on a single server, even if the scheme directs him not to do so.* The client has no way of distinguishing such a single-server case from multi-server storage. She may try to infer via timing [11], but it is not a reliable measure since the inter-server communications are much faster than the client-server communications.

Thus, instead of trying to force the CSP, we give him the flexibility. The CSP may freely employ replication for fault tolerance and availability, and distribution for load balancing and scalability, without the need to inform the client. On the other hand, the client is still ensured that at least one working copy is present, or otherwise the CSP will get caught cheating. Therefore, the CSP is incentivized to make sure he keeps the client's data intact. Our solution does not decrease detection probability of a cheating CSP, while providing improved performance as seen in Section 5.

Version Control. One of the applications of dynamic data outsourcing schemes is outsourced *version control systems* (VCS). Pervasive examples include CVS, SVN, and Git. There have been many efforts to access the old values of updated data being held in data structures. Overmars [29, 30] proposed three methods to search in-the-past. The first method is to build a new structure same as the before, perform the update on the newest copy, and store both of them. This method has a cost of $O(n)$ for both time and space. The second method is to store only the updates. For each query, the corresponding version is built on-the-fly, and the results are generated using it. This method is space-efficient ($O(1)$ per single update), but generating current version takes $O(v)$ time, where v is the total number of the file versions. The third method uses RI-tree to address decomposable search problem, and updates are insertions. The cost for both time and space is $O(1)$.

Driscoll *et al.* [17] studied persistence in data structures considering binary search/red-black trees. They took details of the underlying data structure into account to have $O(\log n)$ query, insertion, and deletion time; and $O(1)$ space bound for insertion and deletion.

Anagnostopoulos *et al.* [1] introduced the notion of *persistent authenticated dictionaries*, where the user can make queries of the type 'Was element e in set S at time t?' and get an authenticated answer, which is used for making authentic statements about the past. The data structure, together with the protocol for queries and updates is called a *persistent authenticated dictionary.* As noted by Anagnostopoulos *et al.* [1], a persistent authenticated dictionary must provide low computational cost, low communication overhead, and high security. They gave a persistent version of an authenticated dictionary based on *skip list* [31] and called it *persistent authenticated skip list.* We will apply persistency on rank-based authenticated skip list to make *persistent rank-based authenticated skip list.*

Erway *et al.* [19] proposed an extension of their DPDP scheme to support version control. If the average number of blocks in a file for each version is n, and there are v versions, their VCS requires $O(\log n + \log v)$ time and space for proofs, whereas our proposal requires only $O(1+\log n)$, which is independent of the number of versions (see [19, 9, 27, 18, 12] for optimality discussion). Furthermore, we show how to combine this VCS with our DR-DPDP to obtain distributed VCS with the same complexity. We also explicitly map VCS operations to provable operations in our DR-DPDP scheme.

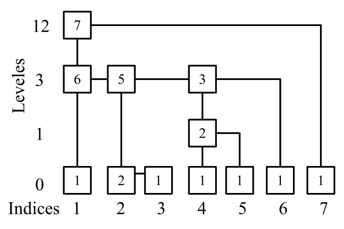
2 Preliminaries

Hash functions are functions that take arbitrary-length strings and output strings of some fixed length, and the goal is to produce random-looking outputs and to avoid collisions. In a hash function H, the *collision* is defined as a pair of distinct inputs x and x', i.e., $x \neq x'$, that is mapped to the same hash values, i.e., $H(x) = H(x')$ [24, 32].

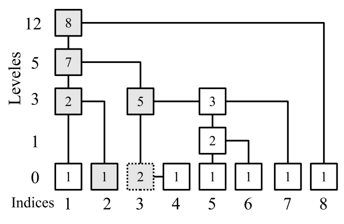
We store the blocks and their corresponding hash values on the server, therefore, we require the hash function to be *collision resistant.* Let $H : \mathcal{K} * \mathcal{M} \rightarrow \mathcal{C}$ be a family of hash functions, identified by $K \in \mathcal{K}$. A hash function is collision resistant if $\forall$ PPT adversaries $\mathcal{A}, \exists$ a negligible function neg such that: $Pr[K \leftarrow \mathcal{K}; (M, M') \leftarrow \mathcal{A}(K) : (M' \neq M) \wedge (H_K(M) = H_K(M'))] \leq neg(n)$. Note that collision resistance implies *second-preimage resistance* [32].

A **skip list** is a randomized data structure, that has binary tree-like properties (i.e., logarithmic operation cost). An **authenticated skip list** provides membership proofs for storing items using a commutative cryptographic hash function h that can be constructed from a collision resistant cryptographic hash function f as: $h(x, y) = f(min(x, y), max(x, y))$. Then, the label of each node in the skip list is computed as the hash of the values of its children. The labels from the queried node to the root make a proof of membership [21]. Merkle hash tree [25] is another widely-used static authenticated data structure.

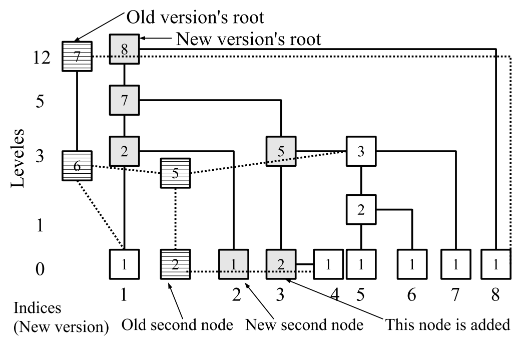
DPDP [19] uses a modified form of the authenticated skip lists called **rank-based authenticated skip list**, where each node v also stores the number of leaf nodes reachable from v (the *rank* of v), as shown in Figure 1a. The file F is divided into n blocks $m_1|m_2|...|m_n$, then a homomorphic tag T_i is computed for each block and put in the skip list, while the blocks are



(a) A rank-based authenticated skip list. Numbers inside the nodes are their ranks.



(b) A node is inserted after the second node. Values on the path up to the root (filled in nodes) are recomputed



(c) A persistent rank-based authenticated skip list, hatched nodes are specific for old version, filled in nodes are specific for new version, and white nodes are in common.

Figure 1: Updating the rank-based authenticated skip list.

stored elsewhere by the server. Each node has two pointers: $rgt(v)$ and $dwn(v)$. Nodes also store a label $f(v)$ that is computed using a collision-resistant hash function h, as a function of $f(rgt(v))$ and $f(dwn(v))$. The client stores locally the label of the skip list's root to verify the membership proofs coming from the server.

An interesting property of this tree-like structure is that the insertion, deletion, or modification of a block affects only the nodes along the path from the block up to the root. The ranks of the affected nodes can be recomputed in constant time per node in a bottom-up way [19]. This is shown in Figure 1b, where a new block is inserted after the second block, at level five.

To make an authenticated skip list *persistent*, the *path-copying* method is applied [1]. A block update results in a new version. The new version consists of all unchanged nodes of the previous version, plus the nodes on the path from the updated block up to the root, whose values are recomputed. Figure 1c shows the process.

DPDP There are two parties in a DPDP scheme: the *client* who wants to off-load her files to the *server*. It consists of the following algorithms as described in [19]:

$KeyGen(1^k) \rightarrow \{sk, pk\}$ is a probabilistic algorithm that the client executes to generate a pair of secret and public keys (sk, pk), given a security parameter as input. The client keeps the secret and public keys, and shares the public key with the server.

$PrepareUpdate(sk, pk, F, info, M_c) \rightarrow \{e(F), e(info), e(M)\}$ The client executes this algorithm to prepare the file to be stored on the server. It takes secret and public keys, the file F, the definition $info$ of the update to be performed, and the previous metadata M_c as input, and generates an encoded version of $e(F)$, the information $e(info)$ about the update, and the new metadata $e(M)$. Finally, the client sends the outputs to the server.

$PerformUpdate(pk, F_{i-1}, M_{i-1}, e(F), e(info), e(M)) \rightarrow \{F_i, M_i, M_c', P_{M_c'}\}$ This algorithm is run by the server when an update request comes from the client. The public key pk, the previous version of the file F_{i-1}, the metadata M_{i-1}, and the outputs of the *PrepareUpdate* function are given as input. It generates the new version of the file F_i, the metadata M_i, along with the metadata and its proof to be sent to the client (M_c' and P_{M_c}').

$VerifyUpdate(sk, pk, F, info, M_c, M_c', P_{M_c'}) \rightarrow \{accept, reject\}$ run by the client with inputs of the *PrepareUpdate* algorithm, M_c' and P_{M_c}' to verify the server's response. It outputs an *acceptance* or *rejection* notification.

$Challenge(sk, pk, M_c) \rightarrow \{c\}$ is a probabilistic algorithm run by the client to create a challenge to be sent to the server. It receives as input the secret and public keys, and the latest client metadata M_c, and generates a challenge c.

$Prove(pk, F_i, M_i, c) \rightarrow \{P\}$ The server runs this algorithm when it receives a challenge from the client, given as input the public key, the latest version of the file and the metadata, and the challenge. It generates a proof P to be sent to the client.

$Verify(sk, pk, M_c, c, P) \rightarrow \{accept, reject\}$ The client runs it after receiving the proof P from the server, giving it the secret and public keys, the client metadata M_c, the challenge, and the proof P as input. The output is an '*accept*' that ideally means that the server keeps storing the file intact, or a '*reject*', otherwise.

For each block i of the file, a tag T_i is generated by the client and sent to the server to be stored. The server stores tags in a rank-based authenticated skip list, but the blocks are stored elsewhere. To check the integrity, the client sends the server a challenge, which is a random list C of blocks to be challenged, together with some random integers r of a fixed length. The server computes a linear combination of challenged blocks $M = \sum_{k \in C} f_k * r_k$ as sum of the random integers multiplied by the blocks. The server also computes an aggregated tag $T = \prod_{k \in C} T_k^{r_k}$ as multiplication of the block tags raised to the power the random integers. Furthermore, the server computes a proof π consisting of all values on the search path from the root of the rank-based authenticated skip list to each challenged block. The whole proof sent to the client will be (M, T, π). For each update, the client sends the required information ($e(F)$ the updated block, and $e(info)$ specifying the update) to the server, who performs the update and returns a proof π that consists of all values on the search path (the block and tag for the update are already known by the client).

3 DR-DPDP

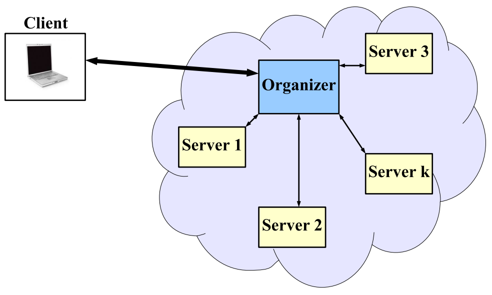
In multi-copy schemes, the organizer combines the results of challenging the same set of blocks on all servers, and returns it as proof to the client. But in distributed case, each server generates a part of the proof, and sends the result to the organizer, who combines them together and returns the result to the client. Having both replication and distribution helps increase availability, reliability, and scalability.

All previous work try to support distribution or replication (or both) of the file, but the problem is that the client has to pre-process the file, and make it ready to be stored on that specific architecture. We extend the DPDP scheme [19] to support distribution and replication of the outsourced file that is *transparent* from the client viewpoint; i.e., the client is *not* required to perform computations dependent on the storage architecture.

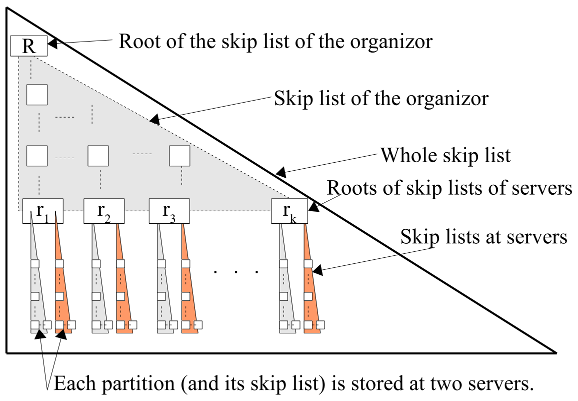
3.1 DR-DPDP Architecture

DR-DPDP is a scheme that provides *transparent* distribution and replication of user data over multiple servers. There are three entities in the model as depicted in Figure 2a. The *client*, who stores data on the CSP, challenges the CSP to check the integrity of data, and updates the stored data. The *organizer*, who is one of the servers in CSP and is responsible for communication with the client and other servers (acts as a gateway or load-balancer). The *servers*, who store the user data, perform provable updates on behalf of the client, and respond to the client challenges coming via the organizer. They only communicate with the organizer and there is no inter-server communication.

It is very important to observe that even though it seems like a central entity, the organizer is not expected to perform any disk operations or expensive group operations (e.g., exponentiation). He will only perform simple hashing, and work with a very small skip list. Hence, his load



(a) The architecture.


R
Root of the skip list of the organizor
Skip list of the organizor
Whole skip list
Roots of skip lists of servers
Skip lists at servers
r1
r2
r3
rk
Each partition (and its skip list) is stored at two servers.

(b) A distributed skip list with 2 replicas.

Figure 2: The DR-DPDP architecture.

will be very light, making it very easy to replicate the organizer to prevent it from becoming a bottleneck or single-point-of-failure. (Further discussion can be found in the Appendix B.)

When the client wants to store a file using this scheme, she first prepares the file as in DPDP, then sends all blocks to the organizer. The organizer divides the file into partitions, each with a predefined number of blocks, and sends each partition to an agreed-upon number of servers (A partition and its rank-based authenticated skip list will be replicated on the specified number of servers.) Each server stores the blocks, builds the corresponding part of the rank-based authenticated skip list, and sends the root value back to the organizer. All servers run in parallel. Once received at least one response for each partition, the organizer builds its own part of the rank-based authenticated skip list and sends the root value as metadata to the client. All these operations are commanded by the organizer and all are transparent to the client.

The idea behind this architecture is that a big rank-based authenticated skip list is divided into multiple sub-lists; the top part is stored on the organizer, and the servers store lower parts, thereby improving scalability. Also, each sub-list will be replicated on a predefined number of servers, improving availability and reliability. Figure 2b shows the idea, where each partition is replicated on two servers. Different servers replicating the same partition are required to use the same randomness to have identical skip lists. Leaves of the organizer's skip list contain the roots of the servers' skip lists, leading to a very small skip list at the organizer.

Remark Note that *single-server DPDP* is a special case of ours, where $R = r_1$ in Figure 2b, and the client and server behavior is unchanged. Moreover, we used a distributed rank-based authenticated skip list to store the tags. With small changes, the 2-3 trees (as used by Zheng and Xu [39]) and Merkle hash trees (as used by Wang *et al.* [36]) can be used instead.

Since the view of DR-DPDP can be seen as a single rank-based authenticated skip list distributed over multiple servers, the format of algorithms are all similar to DPDP algorithms. All algorithms consist of these three steps:

- Find the root of the sub-list containing the requested block(s): This is a usual search operation in skip list performed by the organizer to find leaf node(s) containing information about the server(s) storing the block(s).
- Send the desired command to the server(s): The organizer sends the desired command to those servers. Upon receipt, each server executes the command exactly as in DPDP, on its own skip list, and returns back the results. All servers operate in parallel.
- Construct the result: After receiving the responses from the server(s), the organizer combines together all the results and adds his own rank-based authenticated skip list proof.

3.2 From DPDP to DR-DPDP

This section shows how to use DPDP to construct DR-DPDP. All client operations (KeyGen, PrepareUpdate, VerifyUpdate, Challenge, Verify), and server operations (PerformUpdate, Prove) are the same as DPDP. The organizer operations (PerformUpdate, Prove) are shown in Algorithms 3.1 and 3.2.

Suppose that the file F and a rank-based authenticated skip list on it have been stored in the distributed manner as mentioned. Later, from time to time, the client wants to verify the integrity of the blocks, or update them. The possible updates here, as in DPDP scheme, are insertion of a new block after a given block i, deletion of a block i, and modification of a block i. For all of these operations, the client prepares the desired update command (using PrepareUpdate), and sends it to the organizer, who searches for the block indices in his skip list, figuring out which servers hold which blocks to be updated. Then, he delegates the job to the corresponding servers (All servers holding the same replicas must perform the update.) All servers perform the update in parallel and send the root value to the organizer who picks one proof and metadata per partition among replicas (possible strategies are in the Appendix A), updates his own skip list and sends the new root value to the client (Algorithm 3.1).

Algorithm 3.1: PerformUpdate run by the organizer.

```
   Input: DPDP values sent by the client ($(e(F), e(info), e(M))$).
   Output: DPDP proof to be sent to the client.
1  Interpret $info$ as $\{o_1, o_2, ..., o_l\}$ // list of file block indices to be updated
2  Interpret $e(F)$ as $\{m_{o_1}, m_{o_2}, ..., m_{o_l}\}$ // list of corresponding file blocks
3  $P = \{\}$ // initialize empty proof
4  for $i = 1$ to $l$ do
       // find servers storing the $o_i^{th}$ block from the organizer's skip list
5      $loc_i, \{S_r\}_{r=0}^{s} \leftarrow Search(o_i)$
6      for $j = 1$ to $s$ do
           // Servers perform DPDP update on own partitions, thinking of $F_{i-1}$ as the current version,
           and $M_{i-1}$ as the current skip list root
7          $(M_{c_j}, P_{M_{c_j}}) \leftarrow S_j.PerformUpdate\ (pk, F_{i-1}, M_{i-1}, e(m_{o_i}), e(o_i), e(M))$
       // Pick one proof $P_{M_c}$ and one root $M_c$, how to pick is discussed later
8      $P = P \bigcup P_{M_c}$
   // Put new server roots to the organizer's skip list
9  $(M'_c, P'_{M_c}) \leftarrow PerformUpdate(pk, F_{i-1}, M_{i-1}, \{M_c\}, \{loc_i\}, e(M))$
10 $P = P \bigcup P'_{M_c}$
11 return $M'_c, P$
```

Algorithm 3.2: Prove algorithm run by the organizer.

```
   Input: DPDP challenge sent by the client ($c$).
   Output: DPDP proof to be sent to the client.
   // list of block indices challenged and associated random coefficients
1  Interpret $c$ as $\{o_1, o_2, ..., o_l\}$ and $\{r_1, r_2, ..., r_l\}$
2  $P = \{\}$
3  for $i = 1$ to $l$ do
4      $loc_i, \{S_r\}_{r=0}^{s} \leftarrow Search(o_i)$
       // Select a server from those storing block $o_i$ and challenge it
5      $S_c \in \{S_r\}_{r=0}^{s}$
6      $P_c \leftarrow S_c.Prove(pk, F_i, M_i, c_i)$
7      $P = P \bigcup P_c$
8  return $P$
```

To get an integrity proof, the client generates a challenge command as a list of blocks and random coefficients, and sends it to the organizer. Upon receipt, the organizer finds out which servers hold which blocks, decides on which servers should create the proofs (possibly based on their load), and challenges those servers on the blocks residing in their partition. All servers generate their proofs in parallel, and send them to the organizer. Each proof consists of two parts: a skip list proof, and a combined block. The organizer sums up all combined blocks, and generates the full proof using the sub-proofs and their paths in his own skip list (from the r_is to the R in Figure 2b) as described in Algorithm 3.2.

Frequent insertions or deletions to a partition makes its size very large or small. To solve this problem, repartitioning is required. The repartitioning strategy balances the load on the servers, preserving an amortized time for challenge and update operations (further discussed in the Appendix C).

3.3 Security of DR-DPDP

Since the client-server communication is the same as in DPDP [19], we use the same security definition.

Definition 3.1 (Security of DR-DPDP). *A DR-DPDP scheme is secure if for any PPT adversary who can win the data possession game (from [19]) with non-negligible probability, there exists a polynomial-time extractor that can extract the challenged parts of the file by resetting and challenging the adversary.*

Theorem 3.1. *If DPDP scheme is secure, then our DR-DPDP scheme is secure according to Definition 3.1.*

Proof. All communication between the client and the organizer takes palace as in DPDP. The process is transparent to the client; she thinks as if she communicates with a DPDP server. Moreover, all servers behave as in DPDP. The only difference is how the proof is generated at the organizer, but the resulting proof will be the same as a single-server DPDP proof. Therefore, the organizer-server and inter-server communication is not a matter of security, and rather, we consider the security of client-organizer communication. If the adversary manages to create a verifying proof with non-negligible probability even though all copies of the challenged blocks are corrupted, this means that he managed to cheat either on (at least) one of the server proofs, or the organizer proof. In either case, finally, a DPDP proof is created.

If, at the end of the data possession game [19] the proof is accepted by the challenger with non-negligible probability, then the challenger can extract the requested blocks. The challenger and the extractor we use here are exactly the same as in the DPDP proof, using the 'weighted sums' as described in [19].

Therefore, under the assumption that DPDP is secure, DR-DPDP is secure. The DR-DPDP is as secure as the underlying DPDP in the sense that the client will accept the proof, as long as there is at least one intact copy of her data.

□

Efficiency. When a challenge command comes, the organizer finds the set of servers storing each block, and selects one of them to compute the proof. If the organizer has some knowledge about the servers' current states, e.g., current load of each server, it can perform the selection accordingly, which will affect the overall performance of the system. Also, each server will store and process only a partition of the file, instead of the whole file. Therefore, the DR-DPDP proof generation and update processes are distributed over some servers running in parallel, leading to better performance. This is similar to the shared-nothing architecture that leads to the parallel execution of commands, which in turn will reduce the response time and improve the scalability [22].

Assume each partition has b blocks, and we have p partitions (so $n = pb$ blocks in total). Each server holds a skip list having b leaves. The organizer has a skip list with p leaves. Since all servers run in parallel, the total time complexity of each server's PerformUpdate or Prove functions is $O(\log b)$. The organizer's skip list time is $O(\log p)$, and time for combining proofs is $O(p)$. Since $\log b + \log p = \log n$, the total complexity of DR-DPDP proofs (both computation and communication) is $O(\log n + p)$ for a file with n blocks, regardless of the number of replicas. Note that $p << n$ and mostly even $p \leq \log n$ for realistic values (e.g., $n = 100000$, $p = 10$, $\log n \sim 17$), giving total complexity of $O(\log n)$.

4 Version Control using DPDP

In this section, we show how a persistent rank-based authenticated skip list can be used to build a Version Control System (VCS) like SVN, CVS, Git, etc. We store a file in a persistent rank-based authenticated skip list and assume that each *commit* consists of a series of updates (block modification, deletion, or insertion), resulting in a new version.

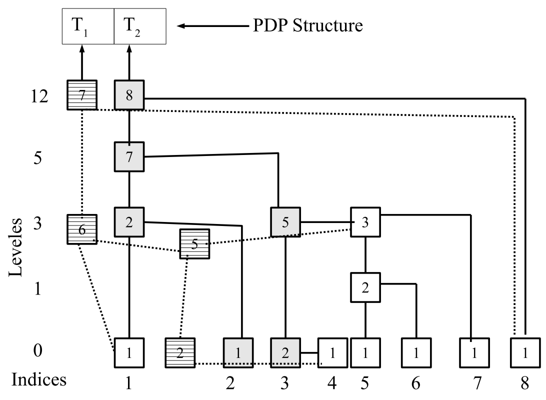


Figure 3: Our VCS architecture.

To manage these versions, Erway *et al.* [19] suggests putting their roots into another rank-based authenticated skip list. But, we use a persistent rank-based authenticated skip list to store the file and its subsequent versions, and put all roots of the persistent skip list into a PDP [2] structure. (Note that a POR scheme [34, 23] can also be employed here, with appropriate algorithm definitions.) Figure 3 presents an instantiation of our VCS. We assume that the client, the organizer, and the servers share a pseudorandom generator seed (or a pseudorandom function key), so that each one can perform any randomized computation independently, while obtaining the same result as the others. The main advantage of this assumption is that, when the client already has a version of the file and performs some updates on it, she can compute the persistent rank-based authenticated skip list root herself, as an honest server would do with the same randomness. She can then compute a PDP tag for that root, and send it to the organizer (or the server in single-server case). The organizer performs the update command, as the client did, and appends the PDP tag to the corresponding PDP structure. This cuts off two rounds from the protocol, eliminating the need for the server to send a DPDP root and proof first, and the client responding back with a tag, thereby improving communication complexity greatly.

An interesting feature of our VCS is that only the content of the first version is explicitly stored on the server. All subsequent versions are stored in terms of the differences between the current and previous versions. These differences are referred to as *deltas* and clearly require much less storage space than the entire contents of a version [28]. In contrast to other schemes which require rebuilding a version from scratch, in our VCS, each version is ready upon request (using persistent rank-based authenticated skip list), solving the problem stated in [17].

4.1 Common Utility Functions

Before describing VCS operations, we present a common utility function to be used in VCS algorithms: $GetVersion(V_i, V_j)$. This algorithm is executed by the client to request the version V_j, when she already holds V_i (which may be null).

- V_i is null or $V_i \geq V_j$: This corresponds to a checkout operation (V_i is null), or to a revert operation ($V_i \geq V_j$). In both cases, the server sends the version V_j from scratch, together with its proof.
- $V_i < V_j$: This corresponds to an update operation, where the client is trying to update to a newer version.
  - If the total number of blocks in version V_j is low compared to the number of changed blocks between V_i and V_j, then it is still better to send all these blocks to the client from scratch (together with their proof).
  - Otherwise, the server sends the differences (*delta*) and their proof separately for each version u such that $V_i < u \leq V_j$.

Normally, the server has to send all deltas starting from the client's current version, one by one, along with their PDP proofs. This requires $O(1 + ed + ed \log n)$ communication, where $d = V_j - V_i$, and e is the average size of deltas. Using the stated trick, we can reduce this complexity to $O(1 + ed)$, since sending only the deltas along with versions' PDP proofs suffices. The client can build the skip list up to the last version using his current blocks and the deltas, and verify the PDP proofs. We separate two cases for proof generation and verification, when the difference is one version ($d = 1$) or multiple versions ($d > 1$):

- **One version**: the server sends the deltas of the new version and the corresponding PDP proof (together with any other information such as commit logs). The client rebuilds the persistent rank-based authenticated skip list, and finds the root. Then, she decides on the validity of the version (by running *PDP Verify* algorithm on the root she computed).
- **Multiple versions**: the server should send the requested blocks, the aggregated PDP proof of all versions, together with all other required information. Now, if the server sends a linear combination of the versions' information, as in PDP, the client has no chance of relating them with individual versions. The client can find by herself, the *fixed-length* part of the Figure 4b, but not the *variable-length* part. If the server sends all versions' information separately, then we loose the $O(1)$ complexity of the PDP proof.

  To solve the problem, the server sends a linear combination of only variable length parts of PDP blocks of requested d versions, achieving $O(1)$ proof size. Let V_{var_k} be the variable-length portion of the PDP block associated with the k^{th} version, V_{fix_k} be the fixed-length portion of length l_{fix}, and r_k be the random challenge sent by the client for version k.

  1. The server computes $V_{var} = \sum_{k=V_i}^{V_j} V_{var_k} * r_k$ and sends to the clientas part of the PDP proof.
  2. After reconstructing persistent rank-based authenticated skip lists, client computes $V_{fix} = \sum_{k=V_i}^{V_j} V_{fix_k} * r_k$ since she now knows each V_{fix_k}.
  3. The client computes $V' = V_{var} * 2^{l_{fix}} + V_{fix}$ by shifting V_{var} to the left l_{fix} times and adding V_{fix}. One can easily verify that V' corresponds exactly to the combined block in a PDP proof. From this point on, the client may perform regular PDP verification using the combined tags received as part of the PDP proof.[1]

[1]We use the version of PDP that does not employ the knowledge-of-exponent assumption and does not take the hash value of the block [2].

4.2 VCS Operations

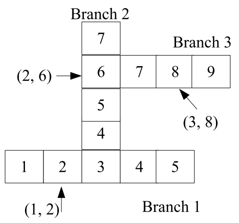
Sink [35] states common functionalities of a VCS as: create, checkout, commit, update, add, edit, delete, diff, revert, log, tag, branch, merge, resolve, and lock. We now show how each of these functionalities are supported by our scheme in a provable manner.

- **Create**: The first upload command issued by the client, creates the repository. One can check if the first version (and hence the repository) is stored by the server, using the common utility functions described above.
- **Update**: The client calls the $GetVersion(V_i, V_j)$ to request the last version V_j from the server and update her *local/working copy*, who is at version V_i.
- **Checkout**: Similar to update with the difference that the client does not have any local copy. She calls $GetVersion(null, V_j)$.
- **Add, edit, delete**: These operations are done locally on the working copy.
- **Diff**: To find the differences between two versions, the server (the organizer in the distributed case) sends the two versions along with their proof to client who can find the differences using a diff algorithm. Alternatively, only deltas with their proofs can be sent.
- **Commit**: After performing all updates on its working copy, the client must commit. Using our above-mentioned trick, the client computes the root of the persistent rank-based authenticated skip list after updates, and a PDP tag for that root. The client sends a DPDP update command with the updated blocks, and a PDP append command for the tag of the new version's root to the server at once. The server(s) update using the above utility functions.
- **Revert**: The client wants to drop what has been changed after some version V_i, and go back to version V_j (possibly $V_j = V_i$). She simply runs $GetVersion(V_i, V_j)$ with the server where V_i is the current version of the client's local copy.
- **Log**: With each commit, the client may provide some logging information (e.g., time and author of the change made, and a summary of changes). The client adds this log to the PDP block related to the version, and builds the PDP tag of the whole block (Figure 4b).
- **Tag**:[2] Name of a branch, can be managed in the same way as 'Log' above.
- **Branch** This operation creates another line of development, and is useful especially in development environments where different groups work on different parts of a project. A version is determined by branch number and version number within the branch. Figure 4a shows a visualization of branching.

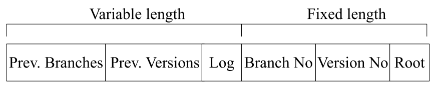
  We store these information about each version: the branch and version number, the root of the corresponding rank-based authenticated skip list, the previous branch that this one was generated from, version of the previous branch that this one has began, the log, and maybe the tag (see Figure 4b).

  Our scheme keeps storage efficient. For each branch, a specific PDP structure will be generated. One may think of one PDP file per branch whose blocks correspond to each version in that branch. But, multiple branches may have many blocks in common, therefore, all share the same blocks.

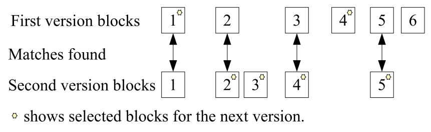
[2]Not to be confused with a PDP tag.



(a) Branching



(b) Information stored in a PDP block.



(c) Matching nodes by the client.

Figure 4: (a)Branching, (b)PDP block structure, and (c)merge.

- **Merge**: This is to combine together two versions of two different/same branches and make a new version in a new/same branch. In development environments, for example, two groups of developers work on their sub-projects separately, and at the end they want to merge what they have done. This operation consists of the following steps: (1) the client requests the two versions of its interest, (2) the server sends those two versions to the client, along with their DPDP and PDP proofs as described in our utility functions, (3) the client runs an algorithm to find and match corresponding nodes of the versions (the skip lists), and then, determines the new version (e.g., Figure 4c) and computes its PDP tag. She then sends all the new version blocks and its PDP tag to the server.
- **Lock**: We believe provably locking something in a client-server setting is a hard (or possibly impossible) problem and consider it out of scope.

4.3 Extensions and Analysis

Multi-client VCS. Our discussion above assumes the same client keeps committing and also retrieving versions. In the single-client case, the client keeps information about the last version, preventing the server from cheating. But, in a multi-client system, the server may cheat and send a previous version –a *replay attack* where the server behaves as if some commit never occurred– to the client (other than the client who created the last version and knows some information about that). The scheme proposed by Erway *et al.* [19] as an extension to DPDP is also vulnerable to this attack. Therefore, some level of inter-client communication is required to prevent the server/organizer from cheating. Each client, after each commit, broadcasts information about her commit to other clients, or puts it on a trusted bulletin board. Just the last version number (and branch number) of the commit needs to be shared between the clients. Sharing of any secret information is *not* necessary (thus the bulletin board can be *public*). We assume the clients trust each other, since they modify the same repository. Now that each client knows the latest version number (of each branch), the server will be caught if he sends a different version.

Distributed VCS. When the client is composed of multiple devices, all connecting to the server to commit or update data, i.e., in software development environments, the above-mentioned central VCS does not suit well, and a distributed VCS (DVCS) is needed.

Using persistent rank-based authenticated skip list, the proposed DR-DPDP scheme can be used to build a DVCS. Each server stores a persistent rank-based authenticated skip list whose roots will be stored in another rank-based authenticated skip list at the organizer. The organizer stores the roots of his own persistent rank-based authenticated skip list (for versions) in the PDP structure. With each update, a new distributed persistent rank-based authenticated skip list with specific updated blocks (the other blocks are shared with the old version) will be built.

The organizer sends the new version's root back to the client. Once the client verified the value of the new root, it computes a PDP tag for the root, and sends it to the organizer for storage. In this case, making commits such a two-step process is important for transparency. Since the CSP architecture should be transparent to the client, she cannot perform the skip list updates locally, even with a shared pseudorandom seed. The client must wait for the organizer to send back the new root before creating the tag. The organizer manages the PDP and communication with the client; hence the distributed architecture is transparent to the client.

Efficiency A proof has two parts: a PDP proof for the version information, and a DPDP proof for the data in that version. The former requires $O(1)$, while the latter needs time and communication complexity $O(\log n)$. The client's storage complexityis $O(1)$, and proof verification complexity is $O(1 + \log n)$ for one version.

4.4 Security of VCS

Definition 4.1 (**Security game for VCS**). *Played between the adversary who acts as a VCS server, and a challenger who plays the role of a VCS client. Full PDP and DPDP game description can be found on the original papers [2, 19]. There are two kinds of VCS commands: update and retrieve. Update commands (i.e., create, commit, branch, and merge) change data on the server, while retrieve commands (i.e., update, checkout, diff, and revert) ask the server to give some parts of the stored files.*

Key generation *The challenger runs the* $KeyGen(1^k) \rightarrow (sk, pk)$*, stores public and private keys (pk, sk), and sends the public key pk to the adversary.*

Query *The adversary specifies an update F and the related information* $info$ *specifying type of the update (e.g., , create, branch, merge), and sends them all to the challenger. The challenger runs* $Commit$ *on them and sends the results to the adversary, who replies with the new metadata and proof, which will be verified by the challenger. The adversary will be notified about the result, and he can repeat this interaction polynomially-many times.*

Setup *The adversary creates a new repository, using the Create command. Then, the above-mentioned interaction is performed again. The challenger updates her local metadata only for the updates whose proofs are accepted.*

Challenge *Let F denote the final version of the file as created by the adversary using the verifying updates in the setup phase. Also, the challenger holds the latest verified metadata. The challenger creates a challenge by picking a random version and running the algorithm* $GetVersion$ *with the adversary, who replies with a proof. The adversary wins if the received proof is accepted.*

Definition 4.2 (**VCS security**). *A VCS scheme is secure if for any PPT adversary who can win the VCS security game with non-negligible probability, there exists a polynomial-time extractor who can extract the challenged version of the file with non-negligible probability by resetting and challenging the adversary.*

Theorem 4.1. *Our VCS (DVCS) is secure according to Definition 4.2, assuming that PDP and DPDP (DR-DPDP) are secure.*

Proof. Both VCS and DVCS work in the same way except that DVCS uses DR-DPDP in the background, so here we only consider the VCS. We already proved that DR-DPDP is secure if DPDP is secure.

A VCS is not secure if the server can prepare proofs accepted by the client when the requested blocks are corrupted, or the blocks used to generate the proof belong to another version. This can be done by creating a DPDP proof, even though (some parts of) the requested challenges do not exist, or a PDP proof, using an old version of the file, to convince the client.

The VCS challenger combines a PDP and a DPDP challenger. She runs the $KeyGen(1^k) \rightarrow (sk, pk)$ which calls the $DPDP.KeyGen(1^k) \rightarrow (sk_{DPDP}, pk_{DPDP})$ and $PDP.KeyGen(1^k) \rightarrow (sk_{PDP}, pk_{PDP})$, sets $sk = (sk_{PDP}, sk_{DPDP})$ and $pk = (pk_{PDP}, pk_{DPDP})$, stores public and private keys (pk, sk), and sends only the public key pk to the adversary..

Whenever the adversary requests a commit, the challenger runs $DPDP.PrepareUpdate$ on the update request from the adversary, and performs $DPDP.PerformUpdate$ locally to find the root of the new version. She computes a PDP tag for this root using $PDP.TagBlock$, and sends the output of $DPDP.PrepareUpdate$ together with the PDP tag to the adversary.

At the challenge phase of the security game, the challenger runs $GetVersion$ on a random challenge. One may think of this as sending a random version number and a series of random blocks in that version (think of this as corresponding to deltas in $GetVersion$). The adversary's response need to include the DPDP root of challenged version, its PDP proof, the challenged blocks of the version, and the DPDP proof of the blocks. The PDP block contains only one data (the DPDP root of challenged version), therefore, can be extracted easily if the PDP proof is accepted (using $PDP.CheckProof$). The extractor simply outputs this data, and is correct since PDP is assumed to be secure. Then, the challenger runs $DPDP.Verify$ to verify the DPDP proof. If it is accepted by the challenger with non-negligible probability, the challenger can extract the requested blocks, again as described in the DPDP security proof [19] solving linear equations.

Therefore, under the assumption that PDP and DPDP (DR-DPDP) are secure, our VCS (DVCS) is secure. □

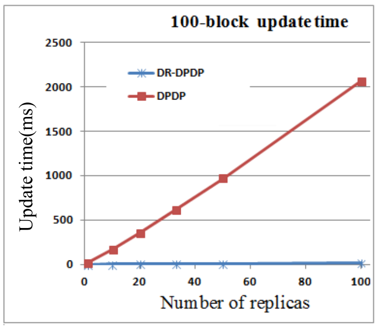
Efficiency. After a file was stored on a server, for each update we store the difference from the previous version, the delta, which needs $\log n$ storage per different block. The storage complexity at the client is $O(1)$, proof generation, communication complexity, and verification are all $O(1 + \log n)$.

5 Performance

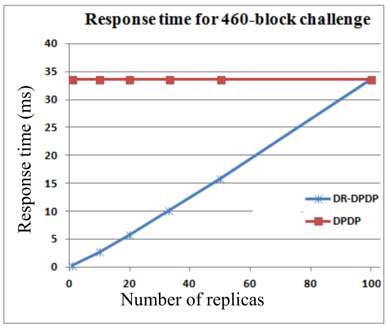
In this section, we compare performance of our DR-DPDP scheme with single-server DPDP. We obtained rank-based authenticated skip list performance numbers from a prototype implementation. All numbers are taken on a regular 2.5GHz machine with 4 cores (but the test running on a single core), with 4GB RAM and Ubuntu 11.10 operating system. The performance numbers are averages from 50 runs. We consider an example scenario with these properties:

- There are 100 servers, and no server stores more than one partition or replica.
- As we increase the number of replicas, the number of partitions will decrease.
- We assume 100000 blocks in total. If each block is 1/2KB, this gives a 50MB VCS (e.g., Tcl CVS repository), while 16KB blocks give a stored file of size 1.6GB. In both cases, it provides a realistic large number.

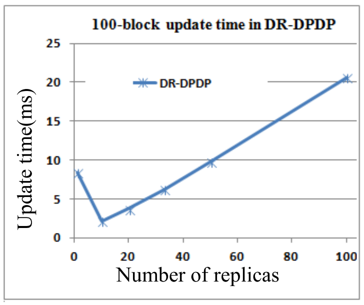
Given this scenario, we compare the server response time for both challenge and update commands, in both (single-server) DPDP and DR-DPDP schemes, and analyze the relation between the number of replicas and the server response time. Note that in DR-DPDP, since the number of servers is fixed, the number of partitions will decrease as the number of replicas increases. Essentially, 100-replica case will have the same challenge-response performance as the single-server case since each server will be storing the whole file in our test.



(a) Update times.



(b) Challenge times.



(c) DR-DPDP update time.

Figure 5: Update and challenge times in DPDP and DR-DPDP.

Figure 5a represents the *total* time taken for a 100-block *update* command in DPDP and DR-DPDP, assuming that the servers in DR-DPDP execute in parallel (except the organizer, who waits for servers' responses first). In single-server DPDP, as the number of replicas grows, the update time will grow linearly, since there is a single server that performs the update on all replicas sequentially. But, in DR-DPDP, the update command is executed by all servers in parallel, and there is no noticeable growth in update time, due to the load balancing property of the distributed scheme.[3]

As the number of replicas grows in DR-DPDP, each server receives challenge commands for a larger number of blocks, therefore the response time will be increased; as shown in Figure 5b. While, in single-server DPDP, the server will select a single replica to respond to challenge commands, and hence, the response time does not depend on the number of replicas. As expected, when all servers store the whole file (100-server 100-replica case), the DR-DPDP performance is equivalent to single-server DPDP, but availability, reliability, and fault-tolerance benefits still do exist (all servers must fail simultaneously for harm to occur).

An interesting property of our proposed scheme is that when the number of replicas is small (the number of partitions is large, and each partition stores a small number of blocks), the size of organizer's rank-based authenticated skip list becomes large. In this case, the computation time in the organizer becomes greater than that of the servers, becoming a bottleneck. Therefore, the total challenge or update time will be large. As the number of replicas grows, the number of partitions falls down, leading to a decrease in the size of the organizer's rank-based authenticated skip list. Since the computation time in the organizer is reduced, the total challenge or update time will decrease. At some point, the total challenge or update time will be minimum, after which the size of each partition becomes large, and hence, the computation time of servers gets large and becomes the bottleneck. Therefore, the total challenge or update time will again increase. This is shown in Figure 5c. Based on the specifications of the underlying hardware, each CSP can determine the optimum number of replicas and partitions (about 10 replicas were the best in our test scenario).

[3]For simplicity, the figure assumes random blocks are updated, resulting in a more-or-less balanced load among the servers.

As for the organizer, consider 10-replica case in the scenario above. This means the organizer's skip list will have only 10 leaves, requiring roughly 0.8KB of memory. Thus, everything the organizer performs can be in memory (a 16GB organizer can store more than *20 million* such skip lists), without requiring disk access. Furthermore, in general it is easy to replicate information that is just 0.8KB in size in real time. These properties render the organizer a viable and attractive option even though it seems to be a centralized entity in the system.

Note that DR-DPDP greatly outperforms DPDP, especially when we realize that network time between the client and the organizer dominates the network time between the organizer and the servers in a typical deployment.

6 Conclusions and Future Work

In this paper, we presented a transparent, distributed, and replicated DPDP. Our scheme extends DPDP to support the distributed architecture of cloud storage. User data is distributed on multiple servers, leading to better scalability, as well as availability and reliability since several servers may store the same partition. Another important feature of our scheme is that the architecture of the cloud storage provider is completely transparent from the client's viewpoint. This fits the real scenario where the CSP wants to hide its architecture, and the client is happy as long as she can reach her file. We also used persistent rank-based authenticated skip list to create a dynamic cloud VCS with optimal complexity ($O(\log n)$). We combined this with our DR-DPDP scheme to obtain a distributed VCS. Our schemes support common VCS operations in a provable manner.

It is interesting to note that some ideas from RAFT [11] may be employed on top of our work. One of the main ideas in RAFT is to correlate the response time of the cloud with the number of hard drives. In our DR-DPDP scheme, it will be related to the number of different servers employed, since each independent server can run in parallel. This way, the client may have an idea about fault tolerance of the system. Yet, we leave such an analysis as future work.

Acknowledgement

We would like to acknowledge the support of TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project numbers 111E019 and 112E115. We also thank Ertem Esiner and Adilet Kachkeev.

References

[1] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. *Information Security*, pages 379–393, 2001.

[2] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *CCS'07*. ACM, 2007.

[3] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm '08*, pages 9:1–9:10. ACM, 2008.

[4] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *Adv.s in Cryptology–ASIACRYPT'09*, pages 319–333, 2009.

[5] A. Barsoum and M. Hasan. Provable possession and replication of data over cloud servers. *CACR, University of Waterloo*, 32, 2010.

[6] A. Barsoum and M. Hasan. Enabling data dynamic and indirect mutual trust for cloud computing storage systems. 2011.

[7] A. Barsoum and M. Hasan. On verifying dynamic multiple data copies over cloud servers. Technical report, Cryptology ePrint Archive, Report 2011/447, 2011.

[8] R. Bhagwan, D. Moore, S. Savage, and G. Voelker. Replication strategies for highly available peer-to-peer storage. *Future Directions in Distributed Computing*, pages 153–158, 2003.

[9] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2):225–244, 1994.

[10] K. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. In *CCS'09*, pages 187–198. ACM, 2009.

[11] K. D. Bowers, M. van Dijk, A. Juels, and A. O. nd Ronald L. Rivest. How to tell if your cloud files are vulnerable to drive crashes. In *CCS'11*. ACM, 2011.

[12] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. Suh. Incremental multiset hash functions and their application to memory integrity checking. In *ASIACRYPT'03*.

[13] T. Cormen, C. Leiserson, R. Rives, and C. Stein. *Introduction to algorithms*. The MIT press, third edition, 2009.

[14] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *ICDCS'08*, pages 411–420. IEEE, 2008.

[15] Y. Deswarte, J. Quisquater, and A. Saïdane. Remote integrity checking. *Integrity and Internal Control in Information Systems VI*, pages 1–11, 2004.

[16] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, 2009.

[17] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[18] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *TCC*, 2009.

[19] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *CCS'09*, pages 213–222. ACM, 2009.

[20] S. Goel and R. Buyya. Data replication strategies in wide area distributed systems. Technical report, ISBN 1-599044181-2, Idea Group Inc., Hershey, PA, USA, 2006.

[21] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Tech. Rep., Johns Hopkins Information Security Institute, 2001.

[22] W. Hasan, D. Florescu, and P. Valduriez. Open issues in parallel query optimization. *SIGMOD Rec.*, 25(3):28–33, 1996.

[23] A. Juels and B. S. Kaliski, Jr. Pors: proofs of retrievability for large files. In *CCS'07*, pages 584–597, New York, NY, USA, 2007. ACM.

[24] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* CHAPMAN & HALL/CRC, 2008.

[25] R. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Security and privacy*, pages 122–134. IEEE Computer Society Press, 1980.

[26] R. Merkle. A digital signature based on a conventional encryption function. *LNCS*, 293:369–378, 1987.

[27] M. Naor and G. Rotblum. Complexity of online memory checking. In *FOCS*, 2005.

[28] B. O'Donovan and J. Grimson. A distributed version control system for wide area networks. *Software Engineering Journal*, 5(5):255–262, 1990.

[29] M. Overmars. Searching in the past i. *RUU-CS*, (81-07), 1981.

[30] M. Overmars. Searching in the past ii: general transforms. *University of Utrecht Dept. of Computer Science Technical Report RUU-CS-81-9*, 1981.

[31] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[32] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, and collision resistance. In *Fast Software Encryption*, pages 371–388. Springer, 2004.

[33] F. Sebé, J. D. Ferrer, A. M. Ballesté, Y. Deswarte, and J. Quisquater. Efficient remote data possession checking in critical information infrastructures. *TKDE'08.*

[34] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology-ASIACRYPT 2008*, pages 90–107. Springer, 2008.

[35] E. Sink. *Version Control by Example.* Pyrenean Gold Press, 1st edition, 2011.

[36] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud. *ESORICS 2009*, pages 355–3700, 2009.

[37] K. Zeng. Publicly verifiable remote data integrity. In *ICICS'08*, pages 419–434, Berlin, Heidelberg, 2008. Springer-Verlag.

[38] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai. Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems. In *HPCC 2010*, pages 434–441, 2010.

[39] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *Proc. of the first ACM conf. on Data and app. security and privacy*, pages 237–248. ACM, 2011.

[40] Y. Zhu, H. Hu, G.-J. Ahn, and M. Yu. Cooperative provable data possession for integrity verification in multi-cloud storage. *IEEE TPDS*, 99(PrePrints), 2012.

[41] Y. Zhu, H. Wang, Z. Hu, G.-J. Ahn, H. Hu, and S. S. Yau. Efficient provable data possession for hybrid clouds. In *CCS'10*, pages 756–758, New York, 2010. ACM.

A Further Analysis of the organizer

There are some trade-offs between efficiency and correctness depending on how the organizer picks one proof and one root among replicas. (1) The organizer can verify the validity of responses from all the servers, and continue only if all the servers got updated correctly. Obviously, this increases the load of the organizer and may introduce delays for the client. Yet, it may be useful for diagnosis purposes at the cloud storage provider, since the organizer can now detect faulty servers and may signal the maintenance team to fix them. (2) The organizer can collect the results and send the first one he received back to the client, without performing any validity check. With this method, the client may face invalid proofs if that one server was faulty. In such a case, the client may ask for a second challenge, to see if there remains at least one intact copy of her data. In response to that second challenge, the organizer may employ the first strategy above instead. (3) As a middle-ground, the organizer may simply compare all the responses for equality (without verifying), and send the proof to the client only if the responses from all replicas match. If there was a mismatch, the organizer may start verifying proofs and use the first one that verifies. The organizer may then again flag the servers with unmatching proofs as faulty.

B Replicating the organizer

The organizer can be replicated to prevent it from being single point of failure. By replication, we gain the advantages *increased availability, increased performance, and enhanced reliability* at the expense of *overhead for creating, maintaining and updating the replicas* [20]. For static schemes, the contents of the organizer's rank-based authenticated skip list can be computed and conveniently stored at all replicas, or may be computed by one organizer and read by all others. But in a dynamic scheme, a replica update protocol is needed; because if the organizer's data is modified by multiple servers, the consistency of the data may be compromised [20].

Goel and Buyya [20] categorized and compared replication strategies in distributed storage and data distribution systems based on the architectural and data management polices used in different systems. Among the four categories defined, the P2P class is suitable for managing replicated organizers.

The granularity at which the data is replicated is an important parameter in replication systems: *whole-data* and *piece-by-piece* [4] [8]. Whole-data replication is simple to implement and has a low state cost, but performs poorly with large data. With piece-by-piece replication the cost to replicate an individual piece can be small, but low availability is an important weakness: the probability of unavailability due to a piece loss is high [8]. To achieve higher availability, erasure correcting codes can be used [8, 20]. Since the size of data stored on the organizer is small, the whole-data replication can be used.

C Repartitioning

If the insertion and deletion operations are uniformly distributed over all partitions, the number of elements in each partition will remain proportional to the number of file blocks, and repartitioning will not be required. However, it is often the case that we do not know in advance what series of operations will be performed on the file. We need a repartitioning strategy to balance the load on servers, preserving an amortized time for challenge and update operations. If some

[4]In the paper they are called whole-file and block-level, respectively.

partitions are more prone to insertion (deletion), after a number of insertions (deletions), resizing is needed. We show that repartitioning does not affect the amortized cost of our scheme, using techniques from [13].

The size of each partition, $2b$, can be determined based on the processing properties of servers, to reach a good average response time. At the beginning, when the client uploads his file, the organizer inserts b blocks sequentially into each partition, reaching a *load factor* $\alpha(T) = 1/2$. We want to bound the load factor between $1/4$ and 1 [13].

After an insertion, if a partition reached the limit of $2b$ blocks, the organizer takes all blocks from that partition, asks the servers storing that partition to delete the stored data, sends them insertion commands for blocks 1 to b, finds a new set of servers, sends them blocks $b+1$ to $2b$ for storing, gets the root of resulting skip lists, and inserts the roots into its own skip list at proper positions.

A server storing b blocks can store another b blocks before repartitioning is needed. The cost of these operations is $O(b \log b)$ in total. Repartitioning, as described above, requires cost b (the cost of building a skip list with b blocks is b, and the two partitions can be built in parallel at all replicas). Hence, the total cost of inserting b blocks is $O(b \log b) + b$. Therefore, the amortized cost for insertion of each block is then $(O(b \log b) + b)/b = O(\log b) + 1 = O(\log b)$ showing that repartitioning does not change the amortized cost for block insertion. Note that we did not take into account the time needed to transfer b blocks from one server to another.

Similarly, the size of a partition may fall down due to multiple deletion operations performed. For load balancing, we bring some blocks from one of its neighbors (read the blocks of the two partitions, delete their skip list, and build two new partitions, each containing half the total blocks read). The cost needed for these operations is at most $(3b/2)O(\log b) + 5b/4$ in total, and $(3/2)O(\log b) + 5/4 \approx O(\log b)$ per block, without taking into account the transfer time of the blocks.

In both cases with repartitioning, the amortized cost of a block insertion or deletion is $O(\log b)$, same as for normal insertion or deletion. So, repartitioning does not affect the amortized cost of these operations, and thus our DR-DPDP scheme can handle amortized logarithmic cost under any workload.