Secure, Dependable and High Performance Cloud Storage

Technical Report UTDCS-51-09
Department of Computer Science
The University of Texas at Dallas
December 2009

Yunqi Ye, I-Ling Yen, Liangliang Xiao, Farokh Bastani, and Bhavani Thuraisingham

Secure, Dependable, and High Performance Cloud Storage

Yunqi Ye, I-Ling Yen, Liangliang Xiao, Farokh Bastani, Bhavani Thuraisingham
Department of Computer Science
University of Texas at Dallas
{yxy078000, ilyen, xll052000, bastani, bhavani.thuraisingham}@utdallas.edu

Abstract

A cloud storage can host a large number of storage servers widely distributed over the Internet, which provides a facilitating environment for achieving high dependability and performance. Added with the short secret sharing (SSS) technology, the security of the data can also be assured. However, there are potential issues in accessing data shares. For example, if the data shares retrieved by a client are inconsistent, the data reconstructed may be invalid. There has been very few works considering access protocols for partitioned data and none considers access semantics for them. In this paper, we analyze the requirements for access protocols in storage systems based on data partitioning techniques to achieve different performance tradeoff, including read, update, as well as the stale version removal algorithms. From these analyses, we formally define the access semantics for data partitioning based storage systems. Then, we develop two access protocols following the access semantics to achieve correct and efficient data accesses. Various protocols are compared experimentally and the results show that our protocols yield much better performance than the existing works.

Keywords: Cloud storage, dependable storage systems, short secret sharing, erasure coding, distributed hash table.

1 Introduction

With technology advances, we are experiencing an information explosion era. Large amount of data are being dynamically created by sensor networks as well as human society daily. The data sources are geographically distributed all over the world. Scientific communities, medical communities, business world, government, and individual users are now relying on these information resources for many advanced tasks. These data demands widely distributed, secure, dependable, and high access performance storage systems and the new infrastructure, cloud storage, has thus emerged [Hadoop, Hay08].

Many current storage systems use storage cluster technologies [Fro03, Abd05]. But due to the locally distributed nature of clusters, they alone are not suitable for the widely distributed storage systems. Cloud storage has the potential of providing geographically distributed storage services since cloud can integrate servers and clusters that are distributed all over the world and offered by different service providers into one virtualized environment. This can potentially resist disastrous failures and achieve low access latency and greatly reduced network traffic by bringing data replicas close to where they are needed. To protect data confidentiality, the replicas are generally encrypted. However, encrypting replicas simply shifts the burden from protecting data to protecting keys. The centralized key management (a single key server or a locally distributed server cluster) is vulnerable to single-point failure [Neu94]. Widely distributed key server systems introduce security risk and, thus, require keys to be encrypted and require another layer of key management. This makes the problem recursive.

An alternative solution for achieving information confidentiality, integrity, and availability is data partitioning. There are several different data partitioning techniques, secret sharing [Sha79], erasure coding [Rab89], and short secret sharing (SSS) [Kra93]. SSS uses erasure codes to generate shares for encrypted data objects and secret share the encryption key. Thus, it has the advantages of erasure coding and secret sharing. In SSS, multiple data shares assure availability. An (n, k) SSS scheme assures security as long as less than k shares are compromised. We have compared various data partitioning schemes and replication with encryption approach and it shows that SSS yields the best overall results [Xia09].

However, a challenge in widely distributed storage systems is directory management. Centralized solutions, such as the name node in [Hadoop], have the potential of single-point failure problem. Some distributed solutions, such as distributed naming servers do not scale up to the cloud environment with global scale. The distributed hash table (DHT) developed for peer-to-peer environment can be used to eliminate the need of directory management and is most scalable in widely distributed systems. DHT can store and locate data and handle the dynamic membership efficiently.

In this paper we consider the design of a secure, dependable, and high access performance cloud storage. We use SSS to protect data from potentially malicious or compromised storage servers and use DHT for share allocation and search. Existing DHT-based P2P storage systems, such as OceanStore [Kub00] and PAST [Dru01], do not specifically consider the DHT design for data shares. In an (n, k) sharing scheme, each client request involves the accesses to at least k data shares. Thus, it is essential to keep a low search cost for each share. Also, in the general DHT schemes, there is no concern regarding which k data shares to access out of the n shares. We design a DHT that is similar to the one-hop-lookup (OHL) [Gup03] approach to minimize the search cost. Our DHT scheme also maintains information to facilitate the identification of near-by shares.

Update is a major issue in data partitioning based approaches. Updating data shares is more complicated than updating data replicas. For replication-based systems without strong consistency requirements, it is fine for the clients to retrieve outdated data. For data partitioning based storage systems, maintaining weak consistency can be an issue. The client must retrieve at least k consistent shares that are of the same version. But concurrent read/update accesses or failures during the update procedure may cause inconsistency among the shares. The clients may read different versions of data shares and the reconstructed data may be invalid (not any prior version of the data). Thus, existing lazy update schemes are not directly usable for SSS.

Some storage systems based on data partitioning approaches consider update protocols [Zha02, Lak03]. Reperasure [Zha02] is based on the DHT and erasure coding scheme. Replicated version files are used to keep track of the update history. Update requests are executed mutual exclusively. For each update, the client reads all the replicated version files to determine a new version number. Then it writes the new version number and *write-in-progress* = *on* to all the version files. After updating the data, the client turns off the *write-in-progress* marker.

Each read operation reads a majority of the version files with *write-in-progress* = *off* and determines the version of the data to be requested from storage servers. This protocol has a high access latency due to the accesses to replicated version files and introduces a large volume of control messages. Furthermore, it cannot handle partial updates due to client failures, which can result in share inconsistency. [Lak03] uses a hybrid of (n, k) verifiable secret sharing and replication schemes and intend to cope with $k-1$ malicious failures. Each share is replicated on m servers. The n shares can be viewed as a row and there are m rows and n columns. The client updates l rows, where $n-(k-1)/l \geq k$. The updated shares are then sent from each of the ln servers to all other servers in the same column, resulting in $O(lnm)$ messages and the new verification strings are sent to all the other servers, resulting in $O(ln^2m)$ messages. A read request accesses $2k-1$ shares to ensure that there are k correct shares. Consistency of the shares can be examined by checking its verification string. The process for disseminating the updated shares and the verification string incurs a high communication overhead among the servers. Since this work considers cluster based storage systems, the overhead is tolerable. However, it cannot scale up to the cloud storage.

To assure correct data reconstruction, we attach a version number to each updated share and reconstruct data from shares with the same version number. Correspondingly, the access protocol should consider how to generate the new version number for update accesses and how to find consistent data shares for read accesses, which impact the system performance significantly. In this paper, we formally define the access semantics for data share accesses, which can be used to guide the design and correctness evaluation of access protocols in data partitioning based storage systems. Though a lot of research works have defined access semantics for replication based storage systems, none of them cover the specific issues in data partitioning based systems. Also, existing works dealing with data share access protocols do not consider access semantics or other formalism. Our attempt in this direction is the first work in the literature.

Based on the access semantics, we consider various version number generation schemes and corresponding access protocols, including the distributed version server (DVS) protocol and the unique version number (UVN) protocol. In DVS, the version number is synchronously generated, which incurs a high update cost (still better than those in existing works) but yields very efficient read accesses. In UVN, the version number is generated by each client independently so that the update cost is low while the read accesses may require more than one round

of message exchange. However, we piggyback the version number retrieval with the read request and response. Compared to [Zha02], UVN can have much better update/read performance and in the worst case, the same read access cost as that in [Zha02].

In DVS, UVN and [Zha02], multiple versions of data are maintained on the storage servers. If the system keeps all the versions, then it will rapidly run into a space problem. Thus, it is necessary to provide mechanisms to remove stale versions for the data shares. In [Zha02], no concrete algorithm is provided for stale version removal. In fact, removing stale version can be a difficult task. Incorrect removal of stale versions of data shares may result in data losses, where no version with at least k shares exists to allow data reconstruction. Also, a malicious node may be able to manipulate the removal process and causes the storage servers delete the data shares aggressively and results in data losses. We develop not only efficient stale version removal algorithms, but also formal semantics for achieving "safe" version removal.

From the results of our experimental studies, we can see that DVS and UVN yield better access performance than [Zha02] as expected. Also, DVS yields better read performance but worse update performance when comparing to UVN.

The rest of the paper is structured as follows. Section 2 introduces the system model and our DHT design. We present the access semantics and protocol requirements in Section 3. Section 4 discusses the DVS and UVN access protocols. The experiment studies and results are given in Section 5. Section 6 states the conclusion of the paper.

2 System Model and DHT Design

We consider "*data object*" as a unit of data stored in the cloud. Let D denote the set of critical data objects. Each data object $d \in D$ has a unique key, which is the search key for the data object. Let $d.data$ denote the actual data of d.

Assume that the cloud consists of N peer storage servers geographically distributed over the Internet. Each storage server can be a single storage platform or a cluster of storage devices. The storage servers are relatively reliable and trustworthy. They may dynamically join or leave the system but at a low rate. Though the servers are

generally trustworthy, some of them may still be malicious or be compromised. Thus, security is still a major concern.

There can be arbitrary clients for the cloud storage. We consider strong access control for client accesses to data objects in D. Different clients may access different subsets of data. Without loss of generality, we consider a single set D that is accessible by a group of clients. These clients are authorized to read and update data objects in D. A malicious client may just write garbage to the data objects that it has the rights to update. But this is beyond the scope of this paper and some approaches, such as [Hu03, Vie05], can be used to detect malicious update requests. So we only consider the fail-stop model for the clients.

To protect the confidentiality of the critical data, we use the (n, k) SSS data partitioning technique to decompose each data object into n shares and distribute them to n storage servers. We index all the secret shares of d for easy accesses. Let $d.share = \{d.sh_i \mid 1 \leq i \leq n\}$ denote the set of shares of d. And let $R_d = \{s_i \mid 1 \leq i \leq n\}$ denote the residence set hosting the shares of d where s_i hosts share $d.sh_i$.

We consider both fail-stop and malicious failures for servers. Generally, (n, k) SSS scheme can tolerate less than k compromised servers and $n-k$ failed servers and still provide data availability and security. So we assume that there are at most f servers failing or being compromised at any time where $f < \min(k, n-k)/2$.

To simplify the directory management in cloud storage, we use the distributed hash table (DHT) scheme to allocate data shares. The network address of each storage server is mapped to a unique m-bit *identifier* (ID) and we assume that the server IDs are uniformly distributed in the ID space. Similarly, the key of each data object is mapped to the same ID space. Let $d.ID$ denote the ID obtained after mapping data d. The ID of a data share of d is given as follows:

$$d.sh_i.ID = (d.ID + i*2^m / n) \% 2^m$$

The data share $d.sh_i$ is assigned to the successor on the ID ring, which is the first physical node on the ring after $d.sh_i.ID$. We use $i*2^m/n$ term to distribute the shares uniformly in the ID space. Though rare, it is still possible that two shares of d are mapped to the same node. In this case, the share with a larger index will be assigned to the successor of the node. Further conflicts are resolved in the same way. Thus, it is assured that no node holds more than one share of the same data object.

In an (n, k) sharing scheme, each client request involves the accesses to at least k data shares. DHT schemes such as Chord [Sto01] and Pastry [Row01] require O(logN) hops for each share lookup. Thus, accessing k shares may incur high access traffic. To resolve this problem, we adopt the one-hop routing approach. Each storage server keeps the full routing table. We also use communication cost estimation mechanisms to help locate near-by shares. Each storage server keeps track of the communication costs to some near-by nodes. In addition, the longitude and latitude coordinates for all the storage servers are maintained in the routing table. This information can be used to calculate the geographical distances and subsequently approximating the communication costs with other nodes when the actual communication costs are not available. The use of the physical location information for communication cost estimation has several benefits. First, the information to be maintained is linear to the number of nodes instead of quadratic. Second, the information can be directly copied from one node to another when new node joins while the actual communication cost cannot be copied. Though previous research has pointed out that the geographic distance does not necessarily correlate to the real communication cost, but [Zha05] has also pointed out that the distance can be used to establish a lower bound on the latency. We have conducted experiments to study the relationships between geographical distance and connection latency and the details are discussed in Section 5.1.

We assume that all the messages are transmitted on a reliable channel, that is, the channel does not change, or duplicate, or drop the messages (there are many existing algorithms to achieve reliable channels). All messages will eventually arrive at the destination unless the destination node fails.

3 Access Protocol Requirements

We use version numbers to resolve the share inconsistency problem. A version number is attached to each share and data can only be reconstructed from the shares with the same version number. However, unlike the version numbers used in conventional storage systems, the data share retrieval and reconstruction makes the version number requirements more complicated.

If only one version of a data share is stored in each server, failure during an update or concurrent updates can bring inconsistent shares which can lead to high read access latency and even read failure. Consider a (10, 4) SSS

scheme. If one update request only manages to update three shares and another manages to update three other shares due to client failures, then the client may have to access all the residence servers to retrieve the four consistent shares. Concurrent accesses may cause the same problem. For concurrent updates, if the shares generated by one update request are overwritten by other updates on some servers while the case is reverse on other servers due to different communication delay among the clients and residence servers, then shares become inconsistent. Also, during the update process, the shares are also inconsistent since some servers may have already received the new shares while others have not. Thus, we should maintain multiple versions of shares of each data object at the residence servers to ensure that the client can find consistent shares from the nearby servers. The multi-version scheme raises some issues and we define new access semantics and consider the corresponding version number requirements in Subsections 3.1 and 3.2, respectively. We discuss how to identify the stale SIs in the residence servers to ensure safe stale SI removal in Subsection 3.3.

3.1 Access Semantics

In replication based distributed storage systems, various consistency models have been proposed to govern the consistency control of data replicas [Ksh08, Ram99]. These consistency models can be used in data partitioning based systems. However, as discussed earlier, in data partitioning based systems with version numbers, we should maintain multiple versions of data shares. Due to the existence of multiple versions on each residence server, one update will not overwrite the other. Thus, the consistency issue in such systems becomes how to choose the "correct" version of shares to return to a read access. What is the "correct" version to be read has different meanings under different consistency models. In a strong consistency (one copy consistency) model, all the accesses are totally ordered and, thus, the "correct" version is clearly defined. When a weaker consistency model is used, the accesses may only have a partial order or even have no clearly defined order. For example, the eventual consistency model only requires that the storage system can converge to a consistent state eventually. In conventional one-version replica based systems, it is clear to return the only version to the read request. However, in a multi-version data partitioning based systems, if the multiple versions do not have clearly defined order, then

which specific version is "correct" to be returned to a read access is also not clearly defined. Here, we attempt to define some reasonable access semantics such that the common effects of single version protocols are achieved.

An update/read access generally involves a sequence of operations, such as the share writing/reading operation or some version number related operations, etc., in some or all of the n residence servers. In order to define the access semantics, we need to first define the order of the accesses, which is given in the following definitions.

Defintion 1. Consider an access a and its operations on a server s_i. Let $a_i = <a_i.o_1, a_i.o_2, a_i.o_3, \ldots>$ denote the sequence of operations of a that are sent to and processed at s_i. For convenience, let $first(a_i)$ denote $a_i.o_1$ and $last(a_i)$ denote $a_i.o_{last}$, where $a_i.o_{last}$ is the last operation of a_i. We define the processing time of the operations as follows.

(i) Let $start(a_i.o_j)$ denote the start time of $a_i.o_j$ and $end(a_i.o_j)$ denote the end time of it.

(ii) If $a_i.o_j$ deos not happen due to client or s_i failure (including malicious failure), or s_i fails after $a_i.o_j$ happens, then $start(a_i.o_j) = end(a_i.o_j) = minTS$. Here, $minTS$ is a conceptual time reference that is less than any other time referenced in the system. Specifically, for any access a in the system, for all i, j, we have $minTS < start(a_i.o_j) < end(a_i.o_j)$ if $start(a_i.o_j) \neq minTS$.

(iii) Given any two operations $a_i.o_j$, and $b_i.o_k$, if $end(a_i.o_j) \leq start(b_i.o_k)$, then $a_i.o_j \rightarrow b_i.o_k$, where $\rightarrow$ denotes the happened before relation.

Definition 2. Consider two update accesses u and v:

(i) If for each i ($1 \leq i \leq n$), $last(u_i) \rightarrow first(v_i)$, and at least one $last(u_i) \neq minTS$, then $u \rightarrow v$.

(ii) If neither $u \rightarrow v$ nor $v \rightarrow u$, then $u \| v$, where $\|$ denotes the concurrent relation.

Definition 3. Consider a read access r and an update u:

(i) Without loss of generality, assume that r contacts a subset of residence servers to retrieve shares. Let S^r denote the set of servers that r contacts. If for each residence server s_i in any one subset of S^r of size k, $end(last(u_i)) \neq minTS$ and $last(u_i) \rightarrow last(r_j)$, then $u \rightarrow r$;

(ii) If for each s_i in S^r, $last(r_i) \rightarrow first(u_i)$, then $r \rightarrow u$;

(iii) If neither $u \rightarrow r$ nor $r \rightarrow u$, then $r \| u$.

For a read access r, the client should get the result of one update from the concurrent updates that are exactly happened before r or the updates that are concurrent with r. The formal requirement of the access semantics is given below.

Requirement 1. Access Semantics. Considering a read access r. If there are two updates u and w such that $w \rightarrow u \rightarrow r$, then r must not retrieve the result of w.

Requirement 1 defines the partial orders of the accesses based on how the operations of the accesses are actually being performed at the residence servers. The goal is to prevent a read access from getting a stale version of data in servers that maintain multiple versions of data shares. Existing consistency models, such as eventual consistency, and strong consistency can be applied in addition to this requirement (though strong consistency already achieves Requirement 1).

3.2 Version Number and Share History

Each version of a data object has an associated version number, which is generated at the creation or update time and attached to each of its shares. The version number consists of a logical timestamp, the ID of the client that issues the update request and the cross checksum. The checksum consists of the hash values of all the shares of a data [Gon89] and is used to assure the integrity of the shares even the compromised storage server modifies the shares and makes them invalid. The version number is attached to each data share of a version of a data object. We define the share instance (SI), which consists of the data share and the version number, as follows.

Definition 4. Let $d.\mathrm{SI}(i, v) = \langle d.sh_i, d.v \rangle$ denote a share instance of a data object d, where $d.sh_i$ is the i-th share of d and $d.v$ is a version number of d.

The purpose of version number is to identify different versions of data shares to allow correct data reconstruction from shares. Thus, the version numbers must satisfy the basic requirement, uniqueness. We first specify this basic requirement formally.

Requirement 2. Uniqueness. Given a data share $d.sh_i$, any two different SIs of it, for instance, $d.\mathrm{SI}(i, v_1)$ and $d.\mathrm{SI}(i, v_2)$, must have different version numbers, i.e., $v_1 \neq v_2$.

As discussed earlier, we need to maintain multiple versions of SIs in the residence servers. In the following, we define the share history formally, which is the sequence of SIs with different version numbers maintained at a residence server.

Definition 5. Let $d.his(i)$ denote the share history for share $d.sh_i$ stored on a server S. $d.his(i)$ is a sequence of SIs ($d.\mathrm{SI}(i, v_1)$, $d.\mathrm{SI}(i, v_2)$, $d.\mathrm{SI}(i, v_3)$, ...) where $d.v_1$, $d.v_2$, $d.v_3$, ... are version numbers of the SIs in $d.his(i)$.

Several issues arise regarding the share history. First, which version of SI in the share history should be returned to the client upon read accesses to result in a set of consistent shares with the same version number as well as to satisfy Requirement 1? Second, how the share history should be maintained so that it facilitates efficient and correct retrieval of the desired version?

Consider a read request. Various approaches can be used for the client to get k consistent shares while satisfying Requirement 1. First, the client can try to get the available version numbers in the share history of the data object on the residence servers before retrieving a specific version of the shares. This always requires two rounds of message exchanges between the client and the servers. Second, server s_i, upon a read request, can return multiple SIs from the top of $d.his(i)$. It is likely for the client to get one consistent version of SIs out of the multiple versions returned from the servers. However, the communication cost can be almost linear to the number of data shares sent. Third, server s_i can return the top SI in $d.his(i)$. If the client receives a sufficient number of consistent shares, then the read access is done in one round of message exchange. Otherwise, the client can request earlier versions of SIs till a set of consistent shares are received. The third approach is an optimistic approach. If the rate of the update accesses is not very high, then it is likely that the client can get consistent shares in one round of message exchange. Or additional rounds of message exchanges will be needed to complete the read access.

In this paper, we use the hybrid of the first and third approaches to avoid the necessity of multiple rounds of retrievals or large size message transmission. In the first round of retrieval, the server s_i returns its version numbers in $d.his(i)$ together with the top SI. If the client does not get a sufficient number of consistent shares, it analyzes the version numbers it receives and finds the highest version number, $d.v_m$, that has a sufficient number of shares to reconstruct the data (we will call $d.v_m$ the designated version). Then in the second round, the client

retrieves the SIs with version number $d.v_m$. In most cases, the read access can complete in one round of retrieval. Otherwise, it will be completed with two rounds of retrievals. Only in the rare cases when some accessed servers fail more than two rounds of retrievals may be necessary.

When using the read access approach discussed above, how to maintain the share history may impact the access semantics as well as the access performance. Consider maintaining the share history in the order of version numbers. If only Requirement 2 is considered, then the access semantics defined in Requirement 1 may be violated. This can be illustrated by the following example: Assume that u and w are the two most recent update accesses on d and there are no other concurrent updates. Also, the version numbers of u and w are $d.v_u$ and $d.v_w$ respectively and $d.v_u > d.v_w$ while $u \rightarrow w$. Then in each server s_i, $d.his(i)$ should contain $<d.\mathrm{SI}(i, v_u), d.\mathrm{SI}(i, v_w), \ldots>$. Thus, the next read access r will read the result from u. But according to Requirement 1, r should get the result from w. Therefore, Requirement 1 is violated. From this example, we can see that without any constraint on the version numbers besides the uniqueness requirement, the version numbers cannot be used to order the share history. And, the only other way to order the share history while satisfying Requirement 1 is to order it by the update time of SIs at each server. In the following, we define the basic requirement for share history ordering.

Requirement 3. Share history ordering by update time. Given a share history $d.his(i)$ at server s_i, the SIs are maintained in the order of the update time of u_i, i.e., $end(last(u_i))$ where u_i is the update access at s_i. Thus the top of $d.his(i)$ is the SI that is most recently written.

However, maintaining share history in the order of the update time of SIs may greatly reduce the probability for the clients to obtain consistent shares in one retrieval round and, hence, significantly impact the access performance. Consider the following example. Assume that a (5, 3) SSS scheme is used and residence server s_i holds $d.SI(i, v)$, for $1 \leq i \leq 5$. Consider three concurrent update accesses to d with version numbers $d.v_1$, $d.v_2$ and $d.v_3$, Assume that due to different communication latencies, the resulting share histories, after the three updates, on servers s_1, s_2 and s_3 are $<d.\mathrm{SI}(1, v_1), d.\mathrm{SI}(1, v_2), d.\mathrm{SI}(1, v_3), \ldots>$, $<d.\mathrm{SI}(2, v_2), d.\mathrm{SI}(2, v_3), d.\mathrm{SI}(2, v_1), \ldots>$ and $<d.\mathrm{SI}(3, v_3), d.\mathrm{SI}(3, v_1), d.\mathrm{SI}(3, v_2), \ldots>$, respectively. Next, a client c contacts residence servers s_1, s_2 and s_3 to get data shares. In the first round, c will get $d.\mathrm{SI}(1, v_1)$, $d.\mathrm{SI}(2, v_2)$ and $d.\mathrm{SI}(3, v_3)$ and they are inconsistent. c has to try another round with a designated version to finish the read access.

The problem discussed above can be resolved if the version numbers are generated in a synchronized manner, that is, the version numbers can reflect the update order. In this case, SIs can be maintained in the order of their version numbers and it is likely that the read access can obtain shares of the same version in one round of share retrieval. Note that It is still possible that the top SIs on the residence servers are not consistent due to client failures during the update or concurrent accesses, but the probability of inconsistency will be much lower. Thus, here we define a stricter requirement which confines the version number generation semantics such that the share history can be ordered by version numbers.

Requirement 4. Version number reflecting update order.

(i) Given two update accesses u_1 and u_2 on the same data object d and their corresponding version numbers $d.v_1$ and $d.v_2$, $u_1 \rightarrow u_2 \Rightarrow d.v_1 < d.v_2$.

(ii) Given a share history $d.his(i)$, the SIs are maintained by their version numbers in decreasing order. Thus the top SI of $d.his(i)$ is the SI that has the maximum version number.

With only Requirement 3, as discussed earlier, it is possible to have a bad read access performance, but it is easier and less costly in version number generation (version numbers only need to be unique) for update requests. When Requirement 4 is satisfied, it is likely that read can be efficient, but synchronized mechanisms will be needed for version number generation. The design choices can be made according to the update access rate. With low rate of concurrent updates, then Requirement 3 will be sufficient.

3.3 Stale Version Removal

Another issue with the share history is the stale version removal (SVR). As the update accesses are issued continuously, there will be more and more SIs with different version numbers for each data object, which may run into a space problem. So it is necessary to remove the stale SIs from the share history. The SVR process must, first of all, be safe such that there will be no data losses due to the removal. Also, it is undesirable that removal of some SIs causes a round of read access failure and thus severe performance degradation.

Let $d.\mathrm{SH}(v_u)$ denote the set of SIs written to the nonfaulty servers by an update u with version number $d.v_u$, i.e. $d.\mathrm{SH}(v_u) = \{d.\mathrm{SI}(i, v_u) \mid 1 \le i \le n \text{ and } s_i \text{ is nonfaulty}\}$. We can use the happened before relation to help identify the

stale versions. For example, given an update access w and its version number $d.v_w$, we can remove all the SIs in $d.\mathrm{SH}(v_w)$ if there is another update u such that $w \rightarrow u$. However, this mechanism has the potential of causing a share retrieval round of a read access to fail. Consider two update accesses u and w and one read access r where w has already completed and u and r are still under execution. Assume that a (5, 3) SSS scheme is used and $S^r = \{s_1, s_2, s_3\}$ where s_i holds $d.SI(i, v)$, for $1 \leq i \leq 3$. Also assume that when r first retrieves the top SIs, the share histories on the three servers are $<d.\mathrm{SI}(1, v_u), d.\mathrm{SI}(1, v_w)>$, $<d.\mathrm{SI}(2, v_w)>$ and $<d.\mathrm{SI}(3, v_w)>$. Thus r receives $d.\mathrm{SI}(1, v_u)$, $d.\mathrm{SI}(2, v_w)$ and $d.\mathrm{SI}(3, v_w)$ and three lists of version numbers from the three servers. Following the read approach discussed in SubSection 3.2, r will try to retrieve $d.\mathrm{SI}(1, v_w)$ in the next round. But if u has completed successfully and the SVR has removed all SIs in $d.\mathrm{SH}(v_w)$ successfully before the second round of retrieval requests of r arrive to the residence servers, then the second round of retrieval of r will fail due to the "aggressive removal". In this case, r may need to request the top SIs again. However, this may result in a racing situation between the read request and the SVR process and, hence, requires many rounds of share retrievals before the read request succeeds.

We consider a conservative stale version removal approach to deal with the aggressive removal problem. If a version $d.v_m$ may be the designated version of a round of retrieval of a read access r, then the version $d.v_m$ should not be removed. Let $\mathrm{SVR}(d.v_m)$ denote the process of removing all the SIs with version number $d.v_m$ from the resident servers. $\mathrm{SVR}(d.v_m)$ consists of several operations, including version check, SI removal, etc. Thus, we can use the notation in Definition 1 for the SVR process. We first define the happened before relation between the SVR processes and read accesses and then give the requirement for the SVR process. Note that if it is not explicitly specified, we assume that all the accesses are on the data object d.

Definition 6. Consider a read access r and an SVR process m:

(i) If r fails, then $m \rightarrow r$.

(ii) If for each residence server s_i in S^r, $last(m_i) \rightarrow first(r_i)$, then $m \rightarrow r$; if for each s_i, $\mathrm{last}(r_i) \rightarrow \mathrm{first}(m_i)$, then $r \rightarrow m$; Otherwise, $m \parallel r$.

Requirement 5. SVR. Consider an SVR process m that tries to remove version $d.v_w$ which is generated by update access w. m should proceed with its removal operation only if:

(i) There is no read access r, such that $m \parallel r$, and there exists another update access u such that $w \rightarrow u$; or

(ii) For any read access r such that $m \parallel r$, there exists another update access u such that $w \rightarrow u \rightarrow r$.

Theorem 1. If Requirement 5 is satisfied, then the aggressive removal problem will not occur.

Proof. We first show that if Requirement 5 is satisfied, there will be no data losses. Consider an update w with version number $d.v_w$. From Requirement 5, before version $d.v_w$ can be removed, there must be another update u such that $w \rightarrow u$. If w is a successful update, that is, w manages to write SIs to all the working residence servers, then from Definition 2, u must also be a successful update (otherwise, we would have $w \parallel u$). Thus, we have that at any time the system has at least one version by a successful update. In other word, it is guaranteed that there is no data loss. Note that for convenience, we consider the initial version of d as the result of the first successful update on d.

Next, we prove that the aggressive removal problem will not occur. Consider an update w with version number $d.v_w$. Assume that $d.v_w$ can be removed in an SVR access m according to Requirement 5. Then for any possible read access r, there are three possible cases.

(1) $r \rightarrow m$. In this case, if r fails, then the considering the aggressive removal problem for r is meaningless. Otherwise, it implies that r is completed before m starts. Thus, aggressive removal problem will not occur.

(2) $m \rightarrow r$. Here since m has completed at all the residence servers in S^r and we have already shown that there is no data loss, i.e., the SIs with version number $d.v_w$ are already removed before r starts. Thus, $d.v_w$ can never become the designated version of r. So, aggressive removal problem will not occur either.

(3) $m \parallel r$. From requirement 5, $d.v_w$ can be removed means that there must exist another update u such that w $u \rightarrow r$. In this case, r should not read the result of w according to Requirement 1. So aggressive removal problem will not occur either. ■

Consider the example given earlier in this section. According to Requirement 5, $d.\mathrm{SH}(v_w)$ cannot be removed even if u has completed successfully since neither (i) or (ii) in Requirement 5 are satisfied.

When Requirement 3 is adopted for share history maintenance, the happened before relation can be determined easily by analyzing the arrival order of SIs in the share histories. However, if Requirement 4, the alternate version of Requirement 3, is used, the actual arrival orders of SIs are lost. In this case, the update order is

determined logically by version numbers. However, directly using the version number order (which is a total order) in place of the happened-before order can cause problem. Consider an example. If we remove $d.\mathrm{SH}(v_w)$ when there is another update u with version number $d.v_u$ such that $d.v_w < d.v_u$, then not only the aggressive removal problem but also the data loss problem may occur, since u can be a partial update and only manages to write less than k SIs successfully.

When the version numbers and share histories are governed by Requirement 4, we can use both the version number order in the share histories and the size of $d.\mathrm{SH}(v_u)$ of each update access u to make SVR decisions. As we have discussed, partial updates are useless and we do not consider them for stale version removal. Given a successful update access u, if u is fully completed, then $|d.\mathrm{SH}(v_u)| = n$. Due to possible server failures, some of the SIs may be missing. But u must write $d.\mathrm{SI}(i, v_u)$ to at least $n-f$ residence servers successfully and result in $n-f \leq |d.\mathrm{SH}(v_u)| < n$. In the worst case, the f failed servers recover to the system and other f residence servers of d fail, then the system still has $n-2f > k$ shares with version number $d.v_u$, which can ensure that the clients can retrieve at least k shares of d with version $d.v_u$ for data reconstruction. In this case, $d.\mathrm{SH}(v_u)$ can be referred to as a *sufficient version set* (SVS). The formal definition of SVS is given below. Based on SVS and version number order we give a variant of Requirement 5 for access mechanisms based on Requirement 4.

Definition 7. Consider an update u with version number $d.v_u$. $d.\mathrm{SH}(v_u)$ is a SVS of d, if it is a subset of $\{d.\mathrm{SI}(i, v_u) \mid 1 \leq i \leq n\}$ and $|d.\mathrm{SH}(v_u)| \geq n-2f$. And we call $d.v_u$ a *stable version number* (SVN) of d.

Requirement 6. SVR. Consider an SVR process m that tries to remove version $d.v_w$ which is generated by update access w. m should proceed with its removal operation only if:

(i) There exists no read access r such that $m \parallel r$ and there exists another version $d.v_u$ such that $d.v_w < d.v_u$ and $d.v_u$ is an SVN; Or

(ii) For any read access r such that $m \parallel r$, there exists another version $d.v_a$ and an update u with version number $d.v_u$ such that $d.v_w < d.v_a \leq d.v_u$, $u \rightarrow r$ and $d.v_a$ is an SVN.

Theorem 2. If Requirement 6 is satisfied, then the aggressive removal problem will not occur.

Proof. First, either (i) or (ii) in Requirement 6 implies that we always keep at least one SVS in the system and, thus, no data loss will occur.

Next, we show that the aggressive removal problem will not happen. Consider a read access r. Similar to the cases discussed in the proof of Theorem 1, if $r \rightarrow m$ or $m \rightarrow r$, the aggressive removal problem will not occur. When $m \parallel r$, from Requirement 6, there exists an SVN $d.v_a$ and an update u with version number $d.v_u$ such that $d.v_w < d.v_a \leq d.v_u$ and $u \rightarrow r$. Since $d.v_a$ is an SVN and $d.v_w < d.v_a$ and the SIs in the share history are maintained by their version numbers in descending order, the SIs of version $d.v_w$ can be the top SIs on at most $2f (< k)$ servers. So r cannot get k consistent shares of version $d.v_w$ when it tries to retrieve the top SIs in the first round of retrieval. Also, because $d.v_w < d.v_u$ and $u \rightarrow r$, r will not select $d.v_w$ as the designated version for next round of retrieval when the version $d.v_u$ exists. Thus, in both cases, removing version $d.v_w$ will not cause the aggressive removal problem. ■

4 Access Protocols

The protocol proposed in [Zha02] does satisfy the access semantics by serializing all updates. But if there are a large amount of update requests, the update delay can be very high due to the waiting time for retrieving the update right. In [Lak03], the update timestamp is generated by each client. So if the clients do not have the synchronized clocks, [Lak03] may violate the access semantics and lose the most recent update because it only keeps one version of each data on servers. We design two protocols, including the distributed version server (DVS) protocol and the unique version number (UVN) protocol. Both of them satisfy the access semantics (Requirement 1) and also support concurrent updates. Also, they satisfy Requirement 2 regarding uniqueness of version numbers.

As discussed in Section 3.2, different designs of the version number generation algorithm can achieve different performance tradeoffs. DVS is designed based on Requirements 4 and 6 to achieve better read performance but incur synchronization costs for generating totally ordered version numbers in update accesses. On the other hand, UVN is designed based on Requirements 3 and 5, which considers partially ordered version numbers to minimize delay in update requests but it may incur read performance penalty.

4.1 Distributed Version Servers Protocol

A simple approach for version number generation is to use a centralized version server that can provide unique and totally ordered version numbers. However, a centralized approach may suffer from the single point failure and performance bottleneck problems. In this section we consider a distributed version server (DVS) protocol. Consider a data object d. In DVS, the version number is maintained by all the residence servers in R_d. Thus, DVS can avoid the single point failure problem. Also, each data object has its own version servers. Thus, the load of version number generation is split, which avoids bottleneck problem. However, comparing to the centralized version server approach in which only two messages are needed, DVS needs $2n$ parallel messages for new version number generation.

4.1.1 Update Algorithm uDVS

Consider the update algorithm in DVS protocol, uDVS. When a client updates a data object d, it first contacts the version servers (residence servers for d) using GETVERSION messages to get logical timestamps. After receiving the responses from all servers (timeout on failed servers is considered as special responses), the client finds *maxTS*, which is the highest timestamp among the responses. Then the client can create a new version number that is greater than any existing one. The pseudo code of algorithm newVN_DVS for new version number generation is given in Figure 1.

```
client.NewVN_DVS(d) {
  foreach server s in R_d
     Send GETVERSION message to s
  Wait until (collect n responses)
  if (#non-timeout responses < n–f)
     return fail
  Compute maxTS
  d.v := <maxTS+1, client.ID, cross-checksum>
  return d.v
}

s_i.processMessage(message){
   if (message.type = GETVERSION)
      return its maximum timestamp in d.his(i)
   ...
}
```

Figure 1. NewVN_DVS algorithm.

Note that in order to get a timestamp that satisfy Requirement 4, it is necessary for the client to contact all the n residence servers. Consider a counter example. Assume that a partial update u with version number $d.v_u$ only

manages to write d.SI(1, v_u) to server s_1 successfully. If a new update w manages to write d.SI(1, v_w) successfully, then, according to Definition 2, $u \rightarrow w$. Thus, from Requirement 4, the protocol should guarantee $d.v_u < d.v_w$. But if w does not request logical timestamp from s_1, then $d.v_w$ may be less than $d.v_u$. On the other hand, if s_1 fails before w can request the timestamp from s_1, then according to Definition 2, $w \parallel u$ and Requirement 4 still holds.

After the new version number is created, the client constructs the n SIs and sends the update request using WRITESHARE messages to the residence servers. Each server maintains an ordered share history for each data object to store the received SIs. Although malicious servers may modify the SIs they received arbitrarily, they cannot affect the SIs on the nonfaulty servers. So the system can keep at least $n-f$ (> k) correct SIs for data reconstruction. The pseudo code for uDVS is given in Figure 2.

```
client.uDVS (d) {
   d.v := client.NewVS_DVS(d)
   {d.sh_1, d.sh_2, ..., d.sh_n} := Encode (d.data)
   foreach i:=1 to n {
      Calculate d.sh_i mapped node s
      d.SI(i, v) := (d.sh_i, d.v)
      Send WRITESHARE message to s
   }
}

s_i.processMessage(message){
   ...
   else if (message.type = WRITESHARE)
        Put d.SI(i, v) to d.his(i)
   ...
}
```

Figure 2. uDVS algorithm.

In the following we will show that with DVS protocol each update has a unique version number (Requirement 2) and the version numbers can reflect the update order (Requirement 4).

Theorem 3. DVS protocol satisfies Requirement 2.

Proof. Consider two updates u and w with version numbers $d.v_u$ and $d.v_w$. If u and w are not concurrent updates, then according to newVN_DVS algorithm, $d.v_u$ and $d.v_w$ have different timestamps so that Requirement 2 holds. If $u \parallel w$, they may get the same *maxTS* and thus, $d.v_u$ and $d.v_w$ have the same timestamp. But they still have different client IDs so that Requirement 2 still holds. Thus, DVS can assure that no two updates will have same the version number. ■

Theorem 4. DVS protocol satisfies Requirement 4.

Proof. To prove that Requirement 4 is satisfied, we first need to show that for any two updates u and w with version numbers $d.v_u$ and $d.v_w$, if $u \rightarrow w$, then $d.v_u < d.v_w$. From Definition 2, if $u \rightarrow w$, then when w requests timestamp on server s_i, u must have already completed writing SI to s_i. Then according to the NewVN_DVS algorithm, w will get a timestamp that is not lower than the timestamp in $d.v_u$ from every residence server that contains $d.\mathrm{SI}(i, v_u)$ and, hence, will obtain a larger timestamp ($maxTS$+1) in $d.v_w$ than $d.v_u$. If a malicious server responds a lower timestamp than its real value, the $maxTS$ obtained by w will not be affected since the nonfaulty servers will give correct responses. And on the other hand, if the malicious server responds a higher timestamp, then the $maxTS$ obtained by w may be even larger and this still follows Requirement 4. Second, according to the uDVS algorithm, all the residence servers maintain an ordered share history for each data object by their version number, which exactly satisfies (ii) of Requirement 4.Thus, Requirement 4 is satisfied by DVS protocol. ■

When Requirement 4 is satisfied, it is likely that read can be efficient, but $2n$ messages are required for version number generations. If the application can tolerate the violations of Requirement 4 for a while, we can only use s_1 in R_d as a leader version server of d for version number generation to reduce the message complexity and improve the performance. The clients also need to track the latest version number of d to verify the responses of the leader during version number generation. If the leader generates a smaller timestamp than the timestamp kept by the client, then the leader is malicious and the version number generation should be switched to DVS protocol. But before the malicious leader is detected, the clients can only assure the uniqueness requirement but Requirement 4 may be violated.

4.1.2 Stale Versions Removal Algorithm SVR_DVS

As discussed in Subsection 3.3, the SVNs are the basis for removing stale versions. Determining SVN requires the knowledge of the update histories of all residence servers. One simple way is to let one lead server gather update histories from all residence servers and determine the SVNs. However, if the lead server is malicious, it may provide false SVNs to the residence servers and lead to incorrect stale version removal (SVR). To tolerate potential malicious failures, we use the residence servers with the first $2f$+1 share indices to determine the SVNs and call them the SVR servers. Let $RS_d = \{svr_i \in R_d \mid 1 \leq i \leq 2f+1\}$ denote the set of SVR servers for

data object d. SVR_DVS determines the SVNs periodically. The duration can be adjusted to balance the tradeoffs between the communication overhead for SVR and the storage space required for keeping the non-stale share history.

Consider a data object d. In the p^{th} period, each residence server s_i constructs its own update history $d.vnl_i^p$ that contains an ordered list of all the version numbers of the SIs received in this period. At the end of the period p, each residence server sends $d.vnl_i^p$ to all the SVR servers through the HISTORY messages. Each SVR server svr_j maintains n ordered version number lists $d.gvnl_i^j$ $(1 \leq i \leq n)$ where $d.gvnl_i^j$ keeps track of the update histories received from residence server s_i. For the p^{th} period, upon receiving $d.vnl_i^p$, the SVR server svr_j merges $d.vnl_i^p$ into $d.gvnl_i^j$. If svr_j gets a time-out when waiting for $d.vnl_i^p$, it simply ignore $d.vnl_i^p$. Then, svr_j constructs an ordered list of SVNs $d.svnl_j$ as follows. It checks each version number in $d.gvnl_i^j$ in ascending order. If a version number $d.v_s$ appears in at least $n-f$ of $d.gvnl_i^j$, then $d.v_s$ will be identified as an SVN and put into $d.svnl_j$. Also, svr_j removes $d.v_s$ from all $d.gvnl_i^j$ to avoid repeated checks in the following periods. In the end, svr_j sends the $d.svnl_j$ to all residence servers. The pseudo code of SVR_DVS algorithm is given in Figure 3.

```
s_i.SVR_DVS(d, p) {
  At end of period {
     Comput RS_d
     Construct d. vnl_i^p
     foreach s in RS_d
       send HISTORY message to s
  }
  Wait until (all servers in RS_d respond or timeout)
  d.bvn_i := max{d.v|d.v is in more than f of d.svnl_j }
  Remove SIs in {d.SI(i, v)|d.v < d.bvn_i} in order
  if(s_i is also SVR server)
     Remove corresponding d.v from all d. gvnl_j^i
}

s_j.processMessage (message) {
  if (message.type = HISTORY){
     Merge d. vnl_i^p into d. gvnl_i^j
     if for i (1≤ i ≤n), d. vnl_i^p timeout or is received{
       foreach version d.v_s in n–f of d. gvnl_i^j){
          Put d.v_s into d.svnl_j
          Remove d.v_s and from all d. gvnl_i^j
       }
       Compute R_d
       foreach s in RS_d
          send d.svnl_j and to s
     }
  }
}
```

Figure 3. SVR_DVS algorithm.

Each residence server s_i will receive at least f+1 responses from the SVR servers, and accordingly, performs the stale version removal. From $d.svnl_j$, for all j, s_i finds the boundary version number $d.bvn_i = \max\{d.v \mid d.v$ is in more than f of $d.svnl_j$ $(1 \le j \le 2f+1)\}$. All SIs in $d.his(i)$ with version numbers less than $d.bvn_i$ are removed in ascending order of their version numbers. In this way, given any two updates w and u such that $w \rightarrow u$, we can ensure that the SIs of w are always removed before the SIs of u. If s_i is also an SVR server, then it also remove this version number from all $d.gvnl_j^i$ $(1 \le j \le n)$.

The stale version removal is performed based on the SVNs. In the following two lemmas, we show that the non-SVN version numbers can never be accepted as SVNs while the version numbers of successful updates (complete without client failure) will be accepted as SVNs in the SVR_DVS algorithm.

Lemma 1. Consider an update u with version number $d.v_u$. If $d.v_u$ is not an SVN, then all nonfaulty SVR servers will not determine $d.v_u$ as an SVN and all nonfaulty residence servers will not accept $d.v_u$ as an SVN.

Proof. Since $d.v_u$ is not an SVN, $|d.\mathrm{SH}(v_u)| < n-2f$ according to Definition 7. So at most $n-2f-1$ nonfaulty residence servers will include $d.v_u$ into their $d.vnl_i^p$. Even if there are f malicious residence servers that also include $d.v_u$ in their $d.vnl_i^p$, $d.v_u$ can only appear in at most $n-f-1$ $d.gvnl_i^j$ on all nonfaulty SVR servers. So $d.v_u$ cannot be determined as an SVN by nonfaulty SVR servers. Thus, the nonfaulty residence servers can only find $d.v_u$ in the responses of at most f servers and will not accept $d.v_u$ as an SVN. ■

Lemma 2. Consider an update u with version number $d.v_u$. If u completes without client failure, then all nonfaulty SVR servers will determine $d.v_u$ as an SVN and all nonfaulty residence servers will accept $d.v_u$ as an SVN.

Proof. Since u completes without client failure, then $|d.\mathrm{SH}(v_u)| \geq n-f$. So at least $n-f$ nonfaulty residence servers will include $d.v_u$ into their $d.vnl_i^p$ (It may take place in different periods for different residence servers). As a result, on all nonfaulty SVR servers, $d.v_u$ will be gathered into at least $n-f$ $d.gvnl_i^j$. Thus all nonfaulty SVR servers will determine it as an SVN no matter malicious servers include $d.v_u$ into $d.vnl_i^p$ or not. So at least $f+1$ SVR servers will put $d.v_u$ into $d.svnl_j$ and send it to the residence servers. Thus all nonfaulty residence servers will find $d.v_u$ in the responses of at least $f+1$ servers and accept $d.v_u$ as an SVN. ■

There are some other version numbers such that for each $d.v_u$ of them, $n-2f \leq |d.\mathrm{SH}(v_u)| < n-f$. According to Definition 7, $d.v_u$ is also an SVN. But the nonfaulty SVR servers may give different determinations according to whether the malicious residence servers include $d.v_u$ into their $d.vnl_i^p$ or not. Thus some residence servers may not accept it as an SVN. But SVR does not require determining all SVNs. So as long as the non-SVN version numbers cannot be accepted as SVNs, the SVR will not be affected.

4.1.3 Read Algorithm rDVS

When a client requests a data object d, it contacts the nearest $k+2f$ residence servers to get their top SIs. Due to the potential temporary inconsistency among the top SIs at different servers, the client should, at the same time, get the version number lists and $d.bvn_i$. This is done with the ReadTop procedure.

However, if the client cannot get k consistent SIs, then it tries to find a version number $d.v_h$, which is the highest version number that appears in at least k version number lists received. If no available version is found,

then $d.v_h = \min_i(d.bvn_i)$. The client then sends $d.v_h$ as the designated version to the servers through READDESI messages.

Upon receiving the READDESI message, a server s_i first compares $d.v_h$ and its current $d.bvn_i$. If $d.v_h < d.bvn_i$, then version $d.bvn_i$ is more desirable for the read access since it corresponds to an update newer than the one of $d.v_h$. Also, since $d.bvn_i$ is an SVN, then on the $k+2f$ contacted servers, at least k consistent SIs of version $d.bvn_i$ should be hosted. Also, in this way, even if $\min_i(d.bvn_i)$ is the designated version $d.v_h$ but it is returned by a malicious server, it cannot cause any problems since we must have $d.v_h < d.bvn_i$. So, s_i just returns the SI with version number $\max(d.v_h, d.bvn_i)$ if $d.his(i)$ contains it. This is done with the ReadDesi procedure. The pseudo code for ReadTop and ReadDesi procedures is given in Figure 4.

```
client.ReadTop(d) {
  foreach s in first k+2f nearest nodes of R_d
    send READTOP message to s
  Wait until {all nodes either respond or timeout}
  {SIs} := responded top SIs
  {VN_Lists} := responded version number lists
  {d.bvn_i} := responded maximum SVNs
  return {{SIs}; {VN_Lists}; {d.bvn_i}}
}

client.ReadDesi (d.v_h) {
  foreach s in first k+2f nearest nodes of R_d
    send READDESI message to s
  Wait until (all nodes either respond or timeout)
  return {SIs in responses}
}

s_i.processMessage (message) {
  ...
  else if (message.type = READTOP)
    return top SI, version numbers in d.his(i) and
      d.bvn_i
  else if (message.type = READDESI){
    d.v_g := max(d.v_h, d.bvn_i)
    if (d.his(i) contains d.SI(i, v_g))
      return d.SI(i, v_g)
  }
  ...
}
```

Figure 4. ReadTop and ReadDesi.

After retrieving at least k consistent shares, the client can reconstruct the data. Besides this, rDVS also assumes the responsibility to repair missed SIs caused by recovered or newly joined servers or partial updates. In this way, the communication cost that the storage servers use to synchronize each other to recover the missed SIs can be reduced. Assume that the client reconstructs the data of version $d.v$. If it detects that a server does not have

the SIs of version $d.v$, it puts the server to set *Recover*. After reconstructing a data object, the client writes back the missing SIs to the residence servers in set *Recover*. The pseudo code of rDVS is given in Figure 5.

```
client.rDVS(d) {
  do {
   {SI_set; VN_lists; {d.bvn_i}} := client.ReadTop(d)
   d.data := client.HandleSI(SI_set)
   if(d.data = NULL){
      Find d.v_h from VN_lists and {d.bvn_i}
      SI_set := client.ReadDesi(d.v_h)
    }
    d.data := client.HandleSI(SI_set)
  }while(d.data = NULL)
  return d.data
}

client.HandleSI(SI_set){
   hashtable := {d.v → {SIs with d.v in SI_set}}
   foreach vs in {hashtable.key} in descent order{
     if(|hashtable.vs| ≥ k)
       return client.Reconstruct(hashtable.vs)
   }
   return NULL
}

client.Reconstruct({SIs with d.v})
  Recover := contacted servers missed version d.v
  d.data := Reconstruct d from {SIs with d.v}
  foreach s in Recover{
    Generate corresponding share d.sh_i
    d.SI(i, v) := (d.sh_i, d.v)
    Send WRITESHARE message to s
  }
  return d.data
}
```

Figure 5. rDVS algorithm.

Though with very low probability, the second round of retrieval may fail due to server failures after the first round of communication or the difference between $d.bvn_i$ on different servers. In this case, the client can redo the read request till a sufficient number of consistent shares are obtained.

4.1.4 Properties of the DVS Protocol

In the following we first give a lemma to show how to determine a happened before relation between an update and a read access conveniently and then show that the DVS protocol satisfies requirements on stale version removal (Requirement 6) and access semantics (Requirement 1).

Lemma 3. Consider a read access r. If r succeeds to retrieve k consistent shares of update u, then $u \rightarrow r$.

Proof. If r succeeds to retrieve k consistent shares of update u, then on at least k servers in S^r, u must complete its last operation, which is writing SI according to the uDVS algorithm, before r retrieves the SI. Thus according to Definition 3, $u \rightarrow r$. ■

Theorem 5. DVS protocol satisfies Requirement 6.

Proof. Consider a read access r and a removal process m on a stale version $d.v_s$. According to the SVR_DVS algorithm, we have $d.v_s < d.bvn_i$. Assume that the version $d.bvn_i$ is generated by update bu. If $r \rightarrow m$ or $m \rightarrow r$, then the removal satisfies (i) in Requirement 6 since $d.bvn_i$ is an SVN and $d.v_s < d.bvn_i$.

Consider the case that $r \parallel m$. If r is in the process of ReadTop procedure, then m does not affect the result of whether r can get k consistent shares since all the SIs of version $d.v_s$ are not top SIs. If r can succeed to get k consistent shares of version $d.v_u$ and assume that the update corresponding to $d.v_u$ is u, we have $u \rightarrow r$ according to Lemma 3. Since the SIs of u are top SIs, $d.bvn_i \leq d.v_u$. Thus, the removal satisfies (ii) in Requirement 6 since $d.v_s < d.bvn_i \leq d.v_u$, $u \rightarrow r$ and $d.bvn_i$ is an SVN.

Then consider the case that r is in the process of ReadDesi procedure and can succeed to get k consistent shares. Assume that $d.v_g = \max(d.v_h, d.bvn_i)$ ($d.v_h$ is the designated version) and update g create the version $d.v_g$. Then it implies that r retrieve k consistent shares of version $d.v_g$ according to the rDVS algorithm so that we have $g \rightarrow r$ according to Lemma 3. Since $d.v_s < d.bvn_i \leq d.v_g$, $g \rightarrow r$ and $d.bvn_i$ is an SVN, the removal still satisfies (ii) in Requirement 6. ■

Theorem 6. DVS protocol satisfies Requirement 1.

Proof. To prove that Requirement 1 is satisfied, we need to show that for any two update accesses u and w with version numbers $d.v_u$ and $d.v_w$, if $w \rightarrow u$, then for any read access r such that $u \rightarrow r$, r should not read version $d.v_w$. Since $w \rightarrow u$, from Theorem 4, we have $d.v_w < d.v_u$ even if there are malicious servers. If the SIs of version $d.v_w$ is already removed, then r cannot read a non-existing version. So Requirement 1 is satisfied. Otherwise, the SIs of version $d.v_u$ cannot be removed either since we remove the SIs in ascending order of their version numbers in the SVR_DVS algorithm.

In the following, we assume that neither of the versions $d.v_u$ and $d.v_w$ is removed. From Definition 2 we have $d.\mathrm{SH}(v_w) \subseteq d.\mathrm{SH}(v_u)$. So if r can retrieve a sufficient number of consistent shares of version $d.v_w$ from residence

servers in S^r, r can also get at least k SIs of version $d.v_u$. Otherwise, we would have $d.\mathrm{SH}(v_w) \not\subseteq d.\mathrm{SH}(v_u)$. Also, because the SIs are maintained in decreasing order of their version numbers in the share history, according to Theorem 4, $d.\mathrm{SI}(i, v_w)$ cannot be the top SI in $d.his(i)$ unless the malicious servers modify the version number of $d.\mathrm{SI}(i, v_w)$. But such modification can be easily detected by checking the cross checksums. And r cannot reconstruct $d.data$ using version $d.v_w$ in the first round of retrieval since at most f SIs of version $d.v_w$ could be the top SIs. Also, $d.v_w$ can never become the designated version before $d.v_u$ according to rDVS algorithm in the second round of share retrieval since on at least k servers in S^r $d.\mathrm{SI}(i, v_u)$ will appear before $d.\mathrm{SI}(i, v_w)$. Thus r can never read version $d.v_w$. ■

4.2 Unique Version Number Protocol

The DVS protocol discussed previously assure a total order of the version numbers and, hence, can improve read performance by increasing the probability of getting at least k consistent shares in the first round of share retrieval. But it also incur extra message exchanges for obtaining a totally ordered version number for update accesses even if the improved leader version server method is used. For applications with a relatively low update rate, concurrent updates occur very rarely and all the SIs are likely to be written to the residence servers in the same order. So it is not necessary to maintain the total order of the version numbers. To reduce the cost, we introduce the Unique Version Number (UVN) protocol, which follows Requirement 1, 2, 3 and 5. In UVN, each client maintains its own logical timestamps and generates the version number by itself. So no communication between the client and the servers is needed for version number generation.

4.2.1 Update Algorithm uUVN

Consider an update to data object d, the client first increments its local timestamp and generates a new version number and correspondingly the new SIs. These SIs are sent to the residence servers and stored in the first position of the share history of each server. In the following we show that with UVN protocol each update will have a unique version number (Requirement 2) and the share history is maintained by the SI arrival order (Requirement 3).

Theorem 7. UVN protocol satisfies Requirement 2.

Proof. Since every time a client generates a new version number, it will increment its logical timestamp, the version numbers generated by one client are unique. At the same time, the version numbers generated by different clients can at least be distinguished by their client IDs so that they are also unique. So UVN assures that no two updates have the same version number. ■

Theorem 8. UVN protocol satisfies Requirement 3.

Proof. In UVN protocol, the residence server always puts the new SI to the first position of the share history. It implies that the share history is maintained in the order that the SIs are received locally and the top SI will be the SI that is most recently written, which exactly satisfies Requirement 3. ■

With the satisfaction of Requirement 3, UVN assures that for two updates u and w, if $w \rightarrow u$, then all SIs of w will appear after the SIs of u in the share histories of nonfaulty servers.

4.2.2 Stale Versions Removal Algorithm SVR_UVN

In UVN protocol, the happened before relations of the updates are preserved by the relative positions of their SIs in the share histories. For example, assume that u and w are two updates with version numbers $d.v_u$ and $d.v_w$. If on each working server s_i, $d.his(i)$ contains <..., d.SI(i, v_w), ..., d.SI(i, v_u), ...> or does not contain d.SI(i, v_u), then $u \rightarrow w$ according to Definition 2 since the update accesses only have one operation (writing SI) on the residence servers in UVN protocol. So by gathering all the version numbers in the share histories and analyzing their orders, we can determine the happened before relations. Similar to the DVS protocol, we use the same set of SVR servers RS_d as that in SVR_DVS algorithm to tolerate possible malicious SVR servers. And also, the SVR_UVN is executed periodically.

In the p^{th} period, each residence server s_i constructs its own update history $d.vnl_i^p$ that contains the version numbers of the SIs received in this period. But unlike the SVR_DVS algorithm, here the version numbers in $d.vnl_i^p$ keep the same order as that of the SIs in the share history. At the end of the period, the residence server sends $d.vnl_i^p$ to all the SVR servers through the HISTORY messages.

Each SVR server svr_j also maintains n version number list $d.gvnl_i^j$ $(1 \le i \le n)$ where$d.gvnl_i^j$ keeps track of the update histories received from residence server s_i. For the p^{th} period, upon receiving $d.vnl_i^p$, the SVR server

svr_j attaches $d.vnl_i^p$ to head of $d.gvnl_i^j$. If svr_j gets a time-out when waiting for $d.vnl_i^p$, it simply ignore $d.vnl_i^p$. Then, svr_j tries to determine the happened before relation as follows. It checks each version number $d.v_w$ in all $d.gvnl_i^j$ from the tail to head and tries to find the first version number $d.v_u$ such that every $d.gvnl_i^j$ contains <..., $d.v_u$, ..., $d.v_w$, ...> unless $d.gvnl_i^j$ does not contain $d.v_w$. Then svr_j records $w \rightarrow u$. In this way, if three updates u, w and y satisfy $w \rightarrow y \rightarrow u$, then $w \rightarrow u$ will not be recorded since it can be inferred from other happened before relations. Sometimes a happened before relation $w \rightarrow u$ recorded in the previous periods may not hold any more because w successfully writes more SIs to the servers. Then the SVR server records $w \nrightarrow u$ to show that this relation should be removed. These records are sent to all the residence servers. The pseudo code of SVR_UVN algorithm is given in Figure 6.

```
s_i.SVR_UVN(d, p) {
  At end of the period {
    Comput RS_d
    Construct d.vnl_i^p
    foreach s in RS_d
      send HISTORY message to s
  }
  Wait until (all servers in RS_d respond or timeout)
  foreach w→u in f+1 responses
    Put w, u and (w, u) into d.UG_i
  foreach w↛u in f+1 responses
    Remove (w, u) from d.UG_i
  bu_i:=first predecessor of updates with SVN sign
       and 0 out-degree
  if(bu_i exists){
    d.SU_i:=subgraph of d.UG_i constructed from
            predecessors of bu_i
    Remove SIs of updates in d.SU_i along edges
    Remove updates in d.SU_i and edges from d.UG_i
    if (s_i is also an SVR server)
      Remove version numbers of updates in d.SU_i
        from all d.gvnl_j^i(1 ≤ j ≤ n)
  }
}

s_j.processMessage (message from s_i) {
  if (message.type = HISTORY){
    Attach d.vnl_i^p to head of d.gvnl_i^j
    if for i (1≤ i ≤n), d.vnl_i^p timeout or is received {
      HBR := ∅
      foreach version d.v_w in d.gvnl_i^j {
        w := update of d.v_w
        if (d.v_w appears in n–f of d.gvnl_i^j )
          Mark w an SVN sign
        if (find the first update u such that w→u)
          HBR := HBR ∪ {w→u}
        if (find previous w→u does not hold)
          HBR := HBR ∪ {w↛u}
      }
      foreach s in R_d
        send HBR to s
    }
  }
  ...
}
```

Figure 6. SVR_UVN algorithm.

It is possible that some updates are still being processed when the residence servers send their $d.vnl_i^p$. Thus these updates may be recognized as partial updates and the SVR server may determine a happened before relation incorrectly. So in SVR_UVN algorithm, when determining the happened before relations, the SVR server also marks the update with an SVN sign if its version number appear in at least $n–f$ lists of $d.gvnl_i^j$. And for a

happened before relation $w \rightarrow u$, only if u has the SVN sign can this relation be used for the stale version removal. As discussed in the SVR_DVS algorithm, such version numbers must be real SVNs.

Each residence server will receive at least f+1 responses from the SVR servers. For each $w \rightarrow u$ or $w \nrightarrow u$, the server accepts it only if it appears in the responses from at least f+1 SVR servers. Similarly, the server accepts the SVN sign of an update u if at least f+1 SVR servers mark u with the SVN sign. But unlike totally ordered version numbers in DVS protocol, the happened before relations only have partial order. So we cannot use just one list to represent all the happened before relations. In SVR_UVN algorithm, each residence server s_i constructs a directed update graph $d.UG_i$ based on the accepted happened before relations following the rules: (1) All updates are the vertices. (2) Given a happened before relation $w \rightarrow u$, $(w, u) \in d.UG_i$. (3) Given a $w \nrightarrow u$, remove (w, u) from $d.UG_i$. Then, s_i first finds all the updates with SVN signs and 0 out-degrees in $d.UG_i$ and then tries to find their first common predecessor with SVN sign as the boundary update bu_i. Let $d.SU_i$ denote the subgraph of $d.UG_i$ that contains all predecessors of bu_i as vertices. Then from the updates with 0 in-degrees in $d.SU_i$ and along with the edges, s_i removes the SI of each update u in $d.SU_i$. In other word, if $w \rightarrow u$, then we remove the SIs of w before that of u. Also, s_i removes u and related edges from $d.UG_i$. And if s_i is also an SVR server, the version number of u is also removed from all $d.gvnl_j^i$ $(1 \leq j \leq n)$. If bu_i does not exist, no shares should be removed.

Here, the happened before relation determination is critical to perform the stale version removal correctly. We first show that UVN will not find happened before relations between concurrent updates.

Lemma 4. Consider two updates u and w with version number $d.v_u$ and $d.v_w$. If $u \| w$, then no nonfaulty SVR servers will determine $u \rightarrow w$ or $w \rightarrow u$ and no nonfaulty residence servers will accept $u \rightarrow w$ or $w \rightarrow u$.

Proof. If $u \| w$, it implies that some nonfaulty residence servers receive u earlier than w so that $d.\mathrm{SI}(i, v_u)$ is after $d.\mathrm{SI}(i, v_w)$ in their share histories while other nonfaulty residence servers receive w earlier than u so that $d.\mathrm{SI}(i, v_w)$ is after $d.\mathrm{SI}(i, v_u)$. So after the SVR servers gathered the version number lists, it is impossible that $d.v_u$ is after $d.v_w$ or $d.v_w$ is after $d.v_u$ in all $d.gvnl_i^j$. Thus no nonfaulty servers will determine $u \rightarrow w$ or $w \rightarrow u$. As a result, the nonfaulty residence servers can only received $u \rightarrow w$ or $w \rightarrow u$ from at most f SVR servers. So they will not accept $u \rightarrow w$ or $w \rightarrow u$.■

The malicious residence servers may provide modified order of version numbers or non-existing version numbers in their $d.vnl_i^p$. Such false information may mislead the SVR servers so that they may not determine any happened before relations in one period and affect the stale version removal. But the following lemma will show that malicious servers cannot block the stale version removal.

Lemma 5. Malicious residence servers cannot block stale version removal.

Proof. Since only the version numbers in the SIs received in one period will be sent out, the malicious servers also have to provide different information for different periods unless they do not provide anything. Otherwise, the malicious servers can be detected by the SVR servers easily.

If no version number are provided by the malicious servers, then the corresponding $d.gvnl_i^j$ on the SVR servers cannot affect the happened before relation determinations according to the SVR_UVN algorithm. Thus the stale version removal will not be affected.

If some non-existing version numbers are provided, then some garbage happened before relations may be found. But these version numbers cannot be identified as SVNs so that they cannot affect the stale version removal.

If some real version numbers are provided, the order of the version numbers may be modified so that some happened before relations are not found. But the relative positions between these version numbers and the version numbers provided in other periods are immutable. Thus, nonfaulty SVR servers can determine happened before relations based on such relative position information correctly. And these happened before relations can be used to perform the stale version removal correctly. So overall, the malicious servers cannot block the stale version removal. ■

4.2.3 Read Algorithm rUVN

Although the share history is not ordered, the read access algorithm rDVS can also be used in the UVN protocol. So for read algorithm rUVN, we also use the same algorithm as rDVS except the following differences.

Upon receiving a READTOP message, rather than the $d.bvn_i$ in rDVS algorithm, a server s_i returns the SVN list that contains the version numbers of the updates with SVN signs and 0 out-degrees in $d.UG_i$ together with the top SI and the version number lists that contains all version numbers in $d.his(i)$.

When the client determines the designated version $d.v_h$, it first tries to find the first version $d.v_a$ that appears in at least k version number lists received from the servers. If no available version is found, then let $d.v_h$ be one SVN that appears in at least k responses. Otherwise, $d.v_h = d.v_a$.

Assume that update h created the designated version $d.v_h$. Upon receiving the READDESI message, a server s_i first checks if there is an update u with version number $d.v_u$ and 0 out-degree in $d.UG_i$ such that $h \rightarrow u$. If u exists, then s_i just returns $d.\mathrm{SI}(i, v_u)$ to the client if $d.his(i)$ contains this SI. While if u does not exist, then s_i returns $d.\mathrm{SI}(i, v_h)$.

4.2.4 Properties of the UVN Protocol

In this section we will show that the UVN protocol satisfies the requirements on stale version removal (Requirement 5) and access semantics (Requirement 1) .

Theorem 9. UVN protocol satisfies Requirement 5.

Proof. Consider a read access r and a removal process m on stale version $d.v_w$. Assume that update w created version $d.v_w$ and the version number of bu_i is $d.v_{bu}$. If $r \rightarrow m$ or $m \rightarrow r$, the removal satisfies (i) in Requirement 5 since $w \rightarrow bu_i$ according to construction method of $d.UG_i$.

Consider the case that $r \parallel m$. According to Definition 6, r will not fail. If r is in process of ReadTop procedure, then m does not affect the result of whether r can get k consistent shares since all the SIs of version $d.v_w$ are not top SIs (Otherwise, $w \rightarrow bu_i$ will not hold). Since there is at least one update u such that $bu_i \rightarrow u$ according to the SVR_UVN algorithm, the SIs of version $d.v_{bu}$ cannot be the top SIs, either. If r can succeed to get k consistent shares from the top SIs, it implies that bu_i writes SIs to at least k residence servers in S^r before r read the top SIs since $d.v_{bu}$ is an SVN and $|S^r| = k+2f$. So we have $bu_i \rightarrow r$ according to Definition 3. As a result, we have $w \rightarrow bu_i \rightarrow r$ and the removal satisfies (ii) in Requirement 5.

Then consider the case that r is in the process of ReadDesi procedure. Assume that the designated version is $d.v_h$ and update h created the version $d.v_h$. From the rUVN algorithm, if r can succeed to get k consistent shares,

then the retrieved version is either $d.v_h$ or a version generated by an update a such that $h \rightarrow a$. Since there is at least one update u such that $bu_i \rightarrow u$, all SIs of bu_i are after the SIs of u in the share history. Thus $d.v_{bu}$ cannot be the designated version. It implies that at least on k servers in S^r, bu_i writes SIs before r retrieves the SIs of a or h. So we have $bu_i \rightarrow r$ according to Definition 3. Thus we have $w \rightarrow bu_i \rightarrow r$ and the removal satisfies (ii) in Requirement 5. ■

Theorem 10. UVN protocol satisfies Requirement 1.

Proof. Similar to theorem 6, to prove that Requirement 1 is satisfied, we need to show that for any two update accesses u and w with version numbers $d.v_u$ and $d.v_w$, if $w \rightarrow u$, then for any read access r such that $u \rightarrow r$, r should not read version $d.v_w$. If the SIs of version $d.v_w$ is already removed, then r cannot read a non-existing version. So Requirement 1 is satisfied. Otherwise, the SIs of version $d.v_u$ cannot be removed either since we remove the SIs following the partial order from the happened before relations in the SVR_DVS algorithm.

In the following, we assume that neither of the versions $d.v_u$ and $d.v_w$ is removed. According to Requirement 3 and Theorem 8, if $w \rightarrow u$, then at all residence server s_i, $1 \leq i \leq n$, $d.his(i)$ should contain <..., d.SI(i, v_u), ..., d.SI(i, v_w), ...> or does not contain d.SI(i, v_w). So the SIs of version $d.v_w$ should not be the top SIs when there are no malicious servers in S^r. Even if some malicious servers may reverse the SI arrival order in their share histories, r cannot reconstruct the data using version $d.v_w$ in the first round of share retrieval since at most f ($< k$) SIs of version $d.v_w$ could be the top SIs. Also, $d.v_w$ cannot be the designated version in the second round of retrieval because on at least k servers the SIs of version $d.v_u$ will be seen by r before the SIs of version $d.v_w$. Thus r can never reconstruct the data of version $d.v_w$ if $u \rightarrow w \rightarrow r$, which exactly satisfies Requirement 1. ■

5 Experimental Studies

We design a simulation system to compare the performance of the access protocols, including DVS, UVN and the protocol proposed in [Zha02] (Let REP denote it). For convenience, let rREP and uREP denote the read and update algorithm of REP respectively. Because REP does not handle partial update, we do not simulate client failures when perform experiments on the REP. The protocol proposed in [Lak03] is designed for a cluster

environment and has a totally different directory management approach. It does not fit for the widely distributed environment so we do not consider it here.

5.1 Experimental Setup

The simulated system consists of 2000 clients and 2000 storage servers hosting 10000 data objects. We apply (n, k) SSS scheme to share the data and use the DHT scheme to distribute data shares to servers. Clients issue data requests to access data objects following a Zipf distribution [Zipf]. We assume that 20 percent of the requests are updates and the others are reads. For partial updates, we assume that the number of shares that are successfully written to the servers follows a uniform distribution between 1 and n. We use read and update access latencies as the metrics for the comparisons.

Each entity (client/server) in the simulated system has a message queue. It takes the message in the queue in the order of arrival and processes it. We assume that it takes 0.1 millisecond to process each message, approximating the disk read time.

Communication is simulated through a global message queue and all the messages sent out by the sender are placed into it. The global message queue takes out the messages in the order of their estimated arrival time and then distributes them into the receivers' message queues for further process. To simulate the realistic communication costs, we conduct experiments on PlanetLab [PLab] platforms and measure the communication latencies between nodes in widely distributed environment. First, we analyzed the correlation between connection latencies (without message load) and geographical distances by measuring the connection costs between 1000 pairs of nodes. A fitting function F1 is derived to describe the relationships, where CL is the connection latency:

F1: $CL = 0.5 + 0.18*(Distance)^{0.74}$

Next, we explore the relationships between the communication latency and message size. We set up socket communication among 1000 communicating pairs and measure the latencies for different message sizes. Based on the results, a fitting function F2 is derived as follows:

F2: $Latency=0.05*(Size)^{0.46}*CL+CL$

In the simulation, the location (longitude and latitude) of each node is randomly generated. Given a message, we first calculate the spherical distance between the sender and receiver from their coordinates. The message size is obtained from the actual implementation of the protocol. From distance and message size, the communication cost is estimated using F1 and F2.

5.2 Experimental Results

We conduct a series of experiments to compare the performance of various access protocols considering different system configurations, including the read/update request arrival rates, client failure rates, and SSS threshold k.

First, we consider different request arrival rates, from 20 to 40000 requests per second. We set $n = 30$, $k = 5$ and the client failure rate to 0.01. The experimental results are shown in Figures 7 and 8. For DVS and UVN, the update access latency stays constant with increasing request arrival rate. But for REP, the update access latency increases as the request arrival rate increases. This is due to the serialized processing of the update requests in REP. For read accesses, as can be seen in Figure 8, the latencies of DVS and UVN increase as request arrival rate increases. This is because higher request arrival rate implies more concurrent read/update accesses, which result in more inconsistent top shares among the residence servers. Thus, more read accesses have to use two rounds of share retrievals to successfully getting k consistent shares. As for REP, since it always requires two rounds of message exchanges for a read request, the read latency is high, but stays constant with increasing arrival rate.

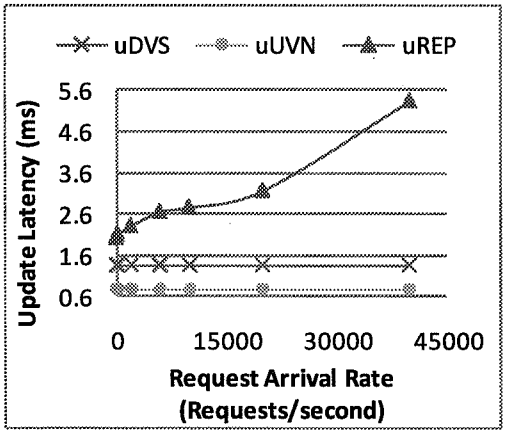

Figure 7. Update latency and request arrival rate

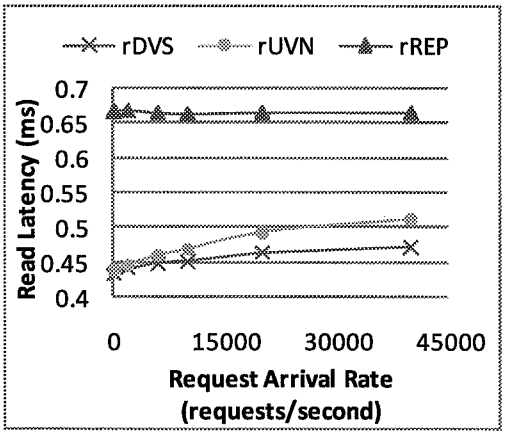

Figure 8. Read latency and request arrival rate

Second, we study the impacts of different client failure rates, from 0.01 to 0.22. We set $n = 30$, $k = 5$ and the request arrival rate to 10000 requests per second. The experimental results are shown in Figures 9 and 10 (REP does not handle client failure and is not considered here). For both DVS and UVN, the client failure rate does not affect the response times of update requests. The read latency increases as the client failure rate increases since client failures result in partial updates which further lead to inconsistency among the top shares on different servers.

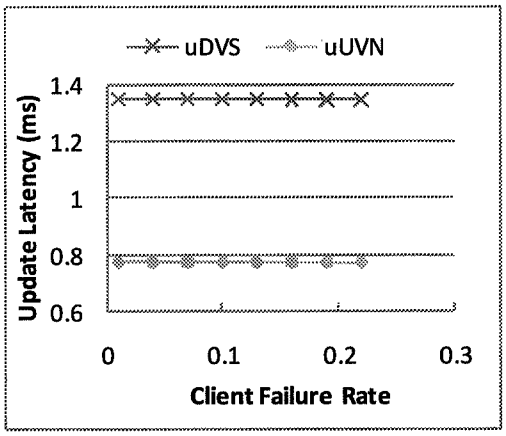

Figure 9. Update latency and client failure rate

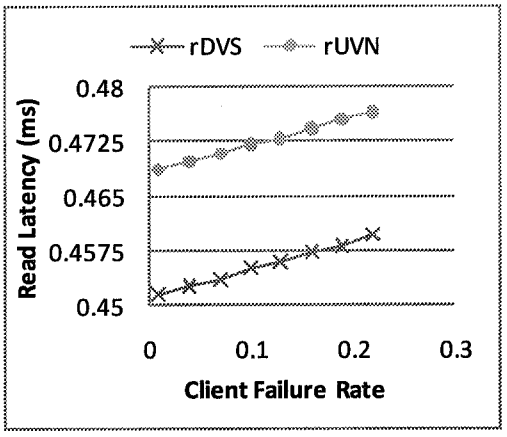

Figure 10. Read latency and client failure rate

Third, we explore the impacts of different threshold k, from 3 to 27. We set $n = 30$, the request arrival rate to 10000, and the client failure rate to 0.01. The experimental results are shown in Figures 11 and 12. As k increases, the share size becomes smaller [Rab89]. So for update accesses, the sizes of WRITESHARE messages become smaller. Thus the update latency decreases as k increases for all the three protocols. But for read accesses, the latency increases as k increases. This is because the clients have to retrieve more shares for data reconstruction and, hence, need to access farther servers, which seems to have a more significant impact than the decreasing message size.

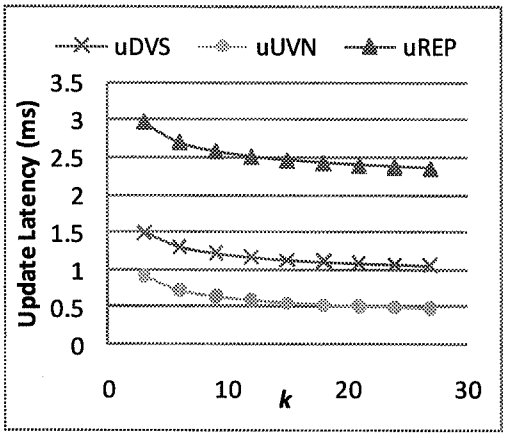

Figure 11. Update latency and *k*

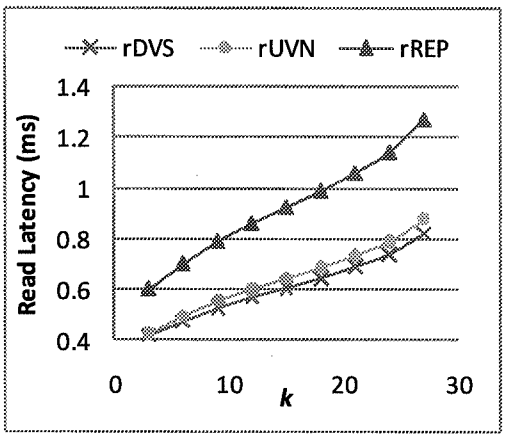

Figure 12. Read latency and *k*

In all the experiments, REP has the worst update and read latencies due to the serialized update request processing and the required two rounds of message exchanges for read accesses. UVN yields best update latencies since the clients can generate new version numbers themselves without contacting any servers, but its read performance is worse than that of DVS since the ordered share histories in DVS lead to a higher chance for read accesses to obtain consistent top shares in one round of share retrieval.

6 Conclusion

In this paper, we consider integrating SSS and DHT technologies to meet the security, dependability and performance requirements of cloud storage. We design a DHT algorithm for share allocation and search. Due to the inconsistency problems with partial updates and concurrent access to shares, we define new access semantics and design new share access protocols. The access semantics defined in this paper can be used in general to guide the design of access algorithms in storage systems with partitioned data (secret sharing or erasure coding). Our access protocols are designed to satisfy the access semenatics and, at the same time, to yield better performance than existing protocols. Experimental results confirm that our algorithms outperform existing approaches.

7 References

[Abd05] M. Abd-El-Malek, W.V. Courtright, C. Cranor, et al. Ursa Minor: Versatile Cluster-based Storage. Proc. of the 4th USENIX Conference on FAST. 2005.

[Dru01] P. Druschel, A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. Proc. of the 8th Workshop on Hot Topics in Operating Systems. 2001.

[Fro03] Svend Frolund, Arif Merchant, Yasushi Saito, et al. FAB: enterprise storage systems on a shoestring. Proc. of the 9th conference on Hot Topics in Operating Systems. 2003.

[Gon89] Li Gong. Securely replicating authentication services. 9th International Conference on Distributed Computing Systems. 1989.

[Gup03] Anjali Gupta, Barbara Liskov, Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. Proc. of the 9th conference on Hot Topics in Operating Systems. 2003.

[Hadoop] Hadoop. http://hadoop.apache.org/core.

[Hay08] Brian Hayes. Cloud computing. Communications of the ACM. 2008.

[Hu03] Yi Hu, Panda, B. Identification of malicious transactions in database systems. Proc. of 7th International Database Engineering and Applications Symposium, 2003.

[Kra93] H. Krawczyk. Secret Sharing Made Short. Crypto'93. 1993.

[Ksh08] Ajay D. Kshemkalyani, Mukesh Singhal. Distributed Computing Principles, Algorithms, and Systems. Cambridge University Press. 2008.

[Kub00] John Kubiatowicz, David Bindel, Yan Chen, et al. OceanStore: An architecture for global-scale persistent storage. Proc. of ACM ASPLOS. 2000.

[Lak03] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored data. IEEE Transactions on Parallel and Distributed systems. 2003.

[Neu94] B. Clifford Neuman, Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. IEEE Communications.1994.

[PLab] http://www.planet-lab.org.

[Rab89] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. Journal of ACM. 1989.

[Ram99] Suchitra Raman, Steven McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. Proc. of the conference on Applications, technologies, architectures, and protocols for computer communication. 1999.

[Row01] Antony Rowstron1, Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. ACM International Conference on Distributed Systems Platforms. 2001

[Sha79] Adi Shamir. How to share a secret. Communications of the ACM. 1979.

[Sto01] Ion Stoica, Robert Morris, David Karger, et al. Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM. 2001.

[Vie05] Marco Vieira, Henrique Madeira. Detection of Malicious Transactions in DBMS. Proc. of 11th Pacific Rim International Symposium on Dependable Computing, 2005.

[Xia09] Liangliang Xiao, Yunqi Ye, I-Ling Yen, Farokh Bastani. Evaluating Dependable Distributed Storage Systems. UTDCS-50-09. 2009.

[Zha02] Zheng Zhang, Qiao Lian. Reperasure: Replication Protocol Using Erasure-Code in Peer-to-Peer Storage Network. Proc. of the 21st IEEE SRDS. 2002.

[Zha05] Hui Zhang, Ashish Goel, Ramesh Govindan. An Empirical Evaluation of Internet Latency Expansion. ACM SIGCOMM Computer Communication Review. 2005.

[Zipf] http://en.wikipedia.org/wiki/Zipf