Abstract

Storage system is an important component in many data intensive applications, including data grid. Security, availability, and high performance are important issues in the storage system design. In this paper we present a Peer-to-Peer (P2P) storage system design based on distributed hash table (DHT) and short secret sharing (SSS) to provide highly available, secure and efficient data storage services. Existing DHT's do not consider share location and search. Also, storage systems using data partitioning schemes (including SSS) does not consider the severe problems in share update. We develop three access protocols to maintain the share consistency in spite of concurrent update, partial update and compromised storage nodes by storing a limit number of history versions of the shares. We also conducted experimental studies to evaluate the performance and data availability and compare the behaviors of the schemes.

Keywords: distributed storage systems, distributed hash table, secret sharing, erasure coding, access protocols.

1 Introduction

Recently, there are many scientific and enterprise applications that deal with a huge amount of data and require data intensive processing. Infrastructures and middleware to support such applications such as data grids and service grids have been widely investigated [Raj03, Aba07, Globus]. One important component in these supporting environments is the storage subsystem. Many of the data intensive applications have high assurance requirements, such as high data availability, security, and high access performance. Storage subsystem design has a determinant factor for these high assurance properties. For example, if the access performance is not enhanced in the storage subsystem, it is difficult to have a high performance data grid. And even if a data grid has the best security policies and access control mechanisms, a compromised storage node can by-pass all those and reveal the critical data. Thus, designing high assurance storage systems is an important task in all data intensive applications.

Many current storage subsystem uses storage cluster technologies [Ghe03, Fro03, Abd05]. A cluster generally consists of a large number of computers or storage nodes that are coupled closely together using a storage area network (SAN). In this type of storage systems, the storage nodes can coordinate easily with each other for data accesses and management. But due to the locally distributed nature of clusters, they are vulnerable to the regional disasters. Also, some clients may be far from the clusters and, hence, incur high access latency. Furthermore, because the nodes in one cluster usually trust each other and even share the same account information, security breach in one node may easily spread to the entire cluster.

In contrast, widely distributed storage systems can resolve the above problems. Geographically distributed storage nodes and clusters from multiple institutions can form a large storage network, which can resist disastrous failures. Also, the clients can access data from near-by storage servers and, hence, can greatly reduce the network traffic and achieve low latency. But a challenge in widely distributed storage systems is directory management. Centralized solutions, such as the single master in GFS [Ghe03] and the object manager in Ursa Minor [Abd05], have the potential single-point failure and performance bottleneck problems. In peer-to-peer (P2P) systems, distributed hash table (DHT) has been proposed to eliminate the need of directory management. DHT can store and locate data and handle the dynamic membership efficiently.

Though widely distributed storage systems offer the potential of better availability and performance than local cluster based systems do, special design is still needed to achieve these properties. A lot of research efforts have been devoted to achieve high assurance storage. Replication is a common technique for improving data availability and access performance [Dou01, Kun03, Row01]. 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) has the same problems as the local cluster based storage systems discussed above [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.

Data partitioning is an alternative solution for achieving high information availability and ensure the data confidentiality and integrity. There are several different data partitioning techniques, secret sharing [Sha79], erasure coding [Rab89], and short secret sharing (SSS) [Kra93]. An (n, k) threshold data partitioning scheme partition the data into n shares and the data can be reconstructed from any k
shares. In secret sharing, no information may leak with less than k shares. Thus, the storage system can maintain confidentiality even if k−1 storage nodes are comprised. However, each secret share has the same size as the original data and, hence, incurs high storage overhead and communication cost. Erasure code has shorter data shares but does not assure that the individual shares do not leak secret information. 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 this paper we consider the design of high assurance widely distributed storage systems. We use DHT for data management and use SSS to protect data from potentially malicious or compromised storage nodes. We consider two issues in the design, the update protocol and the DHT design for data shares. 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. DHT schemes such as Chord [Sto01] and Pastry [Row01] require O(log N) hops for each share lookup. Thus, accessing k shares may incur high access traffic. To resolve this problem, we design a DHT that is similar to the one-hop-lookup (OHL) [Gup03] approach. In the general DHT schemes, there is no consideration on which k data shares to access out of the n shares for a request. Thus, it is likely that the data shares accessed by a request may be located far from the client site. Our DHT scheme is designed to facilitate the accesses of near-by shares.

There are existing storage systems using SSS or other data partitioning schemes, but most of them do not consider the update algorithm [Kub00, Ell05]. Though there is a large body of works considering replica updates [Zha02, Lak03, Agu05, Wyi05], 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 difficult. Concurrent updates 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 no longer be valid (not any prior version of the data). Thus, existing lazy update schemes are not directly usable for SSS.

To assure the share consistency in the reconstructed data, we use the version number to resolve the problems caused by partial updates, concurrent updates, and potentially compromised storage nodes. We present three access protocols based on three different version number implementations: centralized scheme with ordered version number (COVN), distributed scheme with ordered version number (DOVN) and distributed scheme with unique version number (DUVN) together with our DHT scheme. All of the protocols ensure that the clients cannot use inconsistent shares to reconstruct data objects and that the share history cannot increase unlimitedly.

The rest of the paper is structured as follows. Section 2 introduces some related works. The system model is given in Section 3. We present three access protocols in Section 4. Section 5 discusses the experiment study and results. Section 6 states the conclusion of the paper.

2 Related Works

Some storage systems use data fragmentation techniques to achieve better confidentiality and availability. Most of them only consider read accesses or immutable data. For instance, OceanStore only use erasure coding for data archiving. DISP [Ell05] only use erasure coding for immutable data. These systems focus on providing high data availability and do not address the update problems.

Some storage systems based on secret sharing or erasure coding do consider update protocols, but they are mostly cluster-based [Zha02, Lak03, Wyi05]. Thus, the update algorithms rely on the Remote Procedure Call (RPC) protocols. However, this is not applicable in widely distributed systems.

Reperasure [Zha02] integrates the DHT and erasure coding schemes. It considers both read and update operations. A centralized version file is used to keep track of the update history. Concurrent update requests should lock the central version file and, hence, are serialized. Each read operation also needs to read the version file first. This leads to a significant communication overhead and introduces the single point of failures.

[Lak03] proposes a robust and secure distributed storage system using the hybrid of secret sharing and replication technology. The storage servers are divided into n groups and each group stores the $\ell^i$ shares of the data. The servers in the same group are fully replicated. To ensure high confidentiality, the system executes share renewal periodically so that the attacker can never collect enough shares to reconstruct the data despite of a long data life. In the access and share renewal protocol, each server need send shares to other servers in the same group to disseminate the updated or renewed shares and send share verification strings to all the servers to tolerate the compromised servers. So it incurs high volume communication traffic among the servers. Since this work considers cluster based storage systems, the overhead is tolerable. However, it cannot scale up to widely distributed storage systems.

In [Wyi05], the access protocols are based on a quorum system. It uses a total ordered versioning scheme to deal with partial update and concurrent update problems. It assumes that the storage nodes keep all the old versions of the data shares. The read protocol, upon detecting inconsistencies, tries to bring the outdated shares up to date.
However, storing many versions of the shares can be highly space inefficient.

We consider the design of a secure, reliable, and high performance P2P data storage system. In contrast to these existing solutions, we support concurrent updates instead of using lock to serialize them to improve update performance. The clients issue read accesses directly instead of requesting a stable version before read access in order to improve read performance. Our system does not require much inter-server communication so it can scale to widely distributed application environment easily. And we do not store all history versions of shares to reduce space cost since it is not necessary in many applications.

3 System Model

We consider “data object” as a unit of data stored in the system. Let $D$ denote the set of critical data objects in the system. 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 $d$.

The data objects are stored on a set of $N$ storage nodes that are considered as peers and the DHT mechanism is used for object location. When considering a secure system, it is not possible to include arbitrary nodes. Unlike the general P2P storage systems (such as OceanStore), we assume that the nodes in the system have been filtered by non-cyber-world mechanisms and they are relatively trustworthy and stable. A node may leave the system because of failure or routine maintenance and may return after a period of time. Due to the stability of the nodes, the dynamic joining or leaving rate in the system is low. Though the nodes are relatively trustworthy, some nodes may still be malicious or being compromised. Thus, security is still a major concern.

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 nodes. Let $R_d$ denote the residence set hosting the shares of $d$. We index all the secret shares of $d$ for easy accesses. Let $d.s = \{d.s_i | i = 1...n\}$ denote the set of shares of $d$.

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 sent messages will eventually arrive at the destination unless the destination node fails. The failed nodes just stop to provide services while the compromised nodes may behave arbitrarily.

4 DHT for SSS

We use a mapping function such as SHA-1 to map the network address of each storage node to a unique $m$-bit identifier and we assume the node identifiers are uniformly distributed in the identifier space. Similarly, we use the same mapping function to map the key of each data object to an identifier in the same space. Let $d.identifier$ denote the identifier obtained after mapping data $d$. The identifier of a data share of $d$ is given as follows:

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

The data share $d.s$, is assigned to the successor on the identifier ring, which is the first physical node on the ring after $d.s.identifier$. We use $i*2^m/n$ term to distribute the shares uniformly in the identifier 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.

Our DHT uses the concept of one-hop routing where each server keeps the full routing table. One additional feature in our DHT is to support the attempt in accessing near-by shares. Each node in our system kept the longitude and latitude coordinates for all the nodes in the routing table. This information can be used to calculate the geographical distance and subsequently approximating the communication cost. If a node $x$ knows the average communication cost to $y$, then the cost is also kept in the routing table of $x$. This physical location information can be copied from one node to another during node join while the communication cost cannot be copied. We have conducted experiments to study the relationships between geographical distance and ping latency (the details are discussed in Section 6.1). Though previous research such as [Tan04] 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 at the very least. So when there is no other information available, it is reasonable to use geographical distance to approximate communication latency.

5 Access Protocols

As discussed earlier, update accesses to data shares need to be handled carefully. For example, in a replication based storage system, lazy update can be used if only weak consistency is required. But lazy update cannot be used on partitioned data. Consider a $(10, 5)$ SSS scheme. Even with the weak consistency requirement, all 10 shares should be updated in a lock step to avoid reconstruction errors. Only if a version number is used for each share, then the update protocol can be relaxed. Still, it is necessary to update at least 5 shares in a lock step to avoid potential problems. If one client crashes after updating 3 shares and another client crashes after updating another 3 shares, then the data can no longer be reconstructed since we cannot find 5 consistent shares. When reading a data object with weak consistency requirement, we need not always get the latest data. But we must ensure that the shares used to reconstruct the data object are consistent.
We consider three schemes for accesses to the SSS based DHT discussed previously, including COVN, DOVN and DUVN. In all schemes, a version number is attached to each share to ensure correct data reconstruction. Subsection 5.1 discusses the general definition of the version number and related concepts. Subsections 5.2 through 5.5 discuss the access protocols. The read algorithms for the three access protocols are very similar and are discussed in subsection 5.4. Subsection 5.2 considers the update algorithm for COVN, which ensures strong share consistency but may suffer from potential performance bottleneck and single-point failure. In subsection 5.3, we discuss the update algorithm for DOVN, which has the same advantages as COVN but requires a higher number of messages to maintain the version number order. Subsection 5.3 presents the update algorithm for DUVN, which provides better update performance but may require a larger storage space.

5.1 Version Number

The version number consists of a logical timestamp, the identifier of the client (Client ID) that issues the update request and the cross checksum. The logical timestamp together with the client ID make the version number unique and, hence, ensures reconstruction consistency.

Failure during an update can cause the set of shares with a version number “incomplete”, i.e., the number of shares with this specific version number may not be sufficient for reconstructing the original data. Thus, it is necessary to keep multiple versions of a data share to assure that a sufficient number of consistent data shares exist. We call the data share with a specific version number a share instance (SI). Also, we use SI set to refer to the set of SIs with the same version number. If a SI set is generated by a successful update, we call the version number of these SIs a stable version.

A compromised storage node may modify the shares and make them invalid. The cross checksum is used to assure the integrity of the shares. It consists of the hash values of all the shares of a data [Gon89].

5.2 Update Algorithm for COVN

We first consider the centralized approach. The update algorithm for COVN, Update_COVN, is similar to [Zha02]. A dedicated version server is used to generate new version numbers for update requests. A history is maintained at each storage node, ordered by the version number.

For an update request, the client first contacts the version server to get a logical timestamp and then generates the version number for the update. The client then sends the request, including the updated SIs and the version number, to all the residence nodes. The residence node set can be computed using the DHT algorithm. Each storage node, upon receiving the request, stores the SI in its history.

To avoid the accumulation of a large number of SIs, SIs that are no longer useful should be removed. The version server also maintains the recent update information. After a storage node stores an SI, it sends an update success message with the data object key and the version number to the version server. When the version server receives a sufficient number of update success messages and determines that a certain new version is updated completely, it notifies all the residence nodes that the shares with lower version numbers can be removed safely.

Generally, \((n, k)\) SSS scheme can tolerate less than \(k\) compromised nodes and \(n-k\) failed nodes and still provide data availability and security. So we assume that there are at most \(f\) nodes failing or being compromised at any time where \(f < k\) and \(f \leq (n-k)/2\). If \(n-f\) shares are updated successfully, then the update access is successful.

If a client fails before it sends at least \(k\) requests, the newest SIs will not be valid. But because we always have at least one stable version of shares, the data objects is still reconstructable. Thus, partial update will not cause share consistency problem. When there are concurrent updates, the version server can ensure a totally ordered logical timestamp which will result in a consistent history in all storage nodes.

5.3 Update Algorithm for DOVN

In the centralized approach, the version server plays an important role. If it fails, the update access cannot be performed. Though the version server can be replicated, it further increases the communication overhead to ensure consistency among them. Also, if the version server is compromised, it may generate logical timestamp in an incorrect order and result in incorrect data reconstruction. Also, centralized approach may suffer from performance bottleneck. Thus, we consider a distributed approach and introduce the Update_DOVN algorithm. Update_DOVN is similar to the update algorithm in [Wyi05]. The version number is maintained by all the residence nodes of a data object so that current version number information is always available for clients. And the responsibility of removing old SIs is assumed to the clients.

When a client updates a data object \(d\), it contacts the \(k+2f\) nearest residence nodes to retrieve their current version number list. Based on the responses, the client computes maxVersion and maxStableVersion, which are the highest version number and highest stable version number, respectively, among the responses. From maxVersion, the client creates a new version number that is greater than existing ones and sends the update request to the residence nodes. The maxStableVersion is sent back to all storage nodes and the storage nodes can remove all the SIs with version numbers less than maxStableVersion from their history. The pseudo code for Update_DOVN is given in Figure 1.

The version number of the last successful update on data object \(d\) is maxStableVersion. If we only consider server failures without considering partial updates or concurrent updates, then at least \(n-f\) residence nodes have received the new SIs (at most \(f\) node failures). Assume that when an
update request is issued, these \( f \) failed nodes resume but not yet repair the missed SIs while another \( f \) residence nodes fail, then there will be a total of \( 2f \) residence nodes that cannot provoke the client with the latest version number. Therefore, in our algorithm, the client contacts \( k+2f \) residence nodes to retrieve the highest version number.

Since the version number does not have a total order, the history at each storage node is simply stored in a first-in-first-out (FIFO) order. Thus, both partial update and concurrent update may make the histories in the residence nodes inconsistent and subsequently lead to data losses when the storage for histories overflows. But for applications with low concurrent update and failure rates, it is unlikely to have a data object without a stable version and this method can be used to greatly reduce the communication cost.

### 5.5 Read Algorithm

The read algorithms for the three approaches are very similar. So, we first discuss the general concepts and then the differences among the algorithms.

Due to low failure rates in the systems we consider, most of the update requests can be completed successfully. Also, from the analysis in [Fro04], concurrent write-write or read-write operations to the same data object rarely occur. Thus, we consider that with a high probability, the latest SIs in most residence nodes form a stable version. When a client requests a data object \( d \), it first tries to get the latest SIs from the nearest \( k+2f \) residence nodes in parallel. It is very likely for the client to get at least \( k \) consistent shares for the same reason as discussed in Section 5.3.

Besides to reconstruct the data object correctly, the read algorithms also assume the responsibility to repair some missed SIs due to node failures or other reasons. In this way, the communication cost that the storage nodes use to synchronize the SIs can be reduced. When a client determines the consistent shares from the responses, it can also determine which nodes do not have the stable version SIs. Thus after reconstructing a data object, the missing stable SIs on some residence nodes can be inserted into their histories by the client.

Sometimes the SIs obtained from the first round responses may have inconsistent versions due to partial updates or concurrent update with this read operation. If the client cannot find sufficient number of consistent shares for reconstructing the data, then it should get the SIs with an earlier version number. This attempt can continue till a sufficient number of consistent shares are obtained. For the COVN and DOVN schemes, we guarantee that there is at least one stable version of all data objects. So the client can eventually get sufficient consistent shares in several rounds requests. For DUVN scheme, it is very likely that a stable version exists in the system. So in most cases the client also can get sufficient consistent shares in several rounds requests.

We give pseudo code of Read_OVN algorithm for both COVN and DOVN schemes in Figure 2 since the read algorithms for them are almost same. The only difference is the way of processing the WRITESHARE message, which is used for the clients to send new SIs to the residence nodes. In the COVN scheme, this message only
The read algorithm for the DUVN scheme has two main differences from Read_OVN algorithm. First, upon receiving the READLAST message, which is used for the client to request the last version SIs from the residence nodes, the storage nodes respond the SIs in the order that the share is updated locally in the DUVN scheme instead of the order of version number in the ordered version number schemes. Other processes are just same as the Read_OVN algorithm.

6 Experimental Studies

6.1 Simulation Model

A simulator was built to evaluate and compare the performance of the COVN, DOVN and DUVN access protocols. We consider different $k$ and $n$ values and number of storage nodes. In the default configuration, $k=3$, $n=29$ and the system has 2000 storage nodes.

Different performance metrics have been used to evaluate storage access algorithms. [Lak03] uses the number of storage servers to be accessed for each request as the performance indicator. Similarly, [Fro04] uses the number of messages and consumed bandwidth as the performance metrics. These systems are local cluster based and, hence, the latencies for the messages from a client to the nodes in the cluster are almost the same. For P2P systems, these metrics cannot properly reflect the real request processing latency. Many DHT schemes, such as Chord and OHL, use the number of hops at the overlay network layer to represent the communication cost. However, the communication cost for different “hops” can differ significantly. Exiting network simulators, such as ns-2 [Ns2], support the investigation of various network layer protocols, but do not provide realistic traffic data for high level simulations. Topology generators, such as Inet [Inet3], can be used to simulate network topologies to support application layer performance studies. However, these models only support the hop counts (in the raw network layer) while experimental study shows that the actual communication costs for different hops may be quite different.

We develop a more realistic simulation model to simulate communication costs between nodes in P2P environment. We study the relations between communication cost and geographical distance and message size and use the results to build the simulation model.

6.1.1 Generation of Geographically Distributed Nodes

We generate simulated nodes on a globe. We use the longitude and latitude to describe the locations of the nodes. By generating a longitude coordination in [-180, 180] and a latitude coordination in [-90, 90], we get a storage node in the globe. The geographical distance is defined as the spherical distance between two nodes so that we can use F1 and F2 (in subsection 6.1.2) to predict the message latency.

6.1.2 Communication Latency

We have analyzed the correlation between ping latencies and geographical distances. We measure ping cost between

```
Read_OVN(d) {
    Calculate $R_d$
    \{SIs; version\} := ReadLast (d)
    d.data := Reconstruct $d$ from \{SIs\}
    foreach s in the (k+2f) nearest nodes of $R_d$
        if (s is marked REPAIR)
            Generate corresponding share $d.s_i$
            SI := (d.s_i, version)
            Send WRITESHARE message to $s$
    return d.data
}
ReadShare(d) {
    repeat {
        foreach s in the k+2f nearest nodes of $R_d$
            send READLAST message to $s$
    } until (collect k+2 responses)
    \{SIs\} := responded shares
    repeat {
        if (one version num in \{SIs\} = k+f)
            return \{SIs; version\}
        else if (one version num in \{SIs\} ≥ k)
            Mark nodes missing version as REPAIR
            return \{SIs; version\}
    }
    else
        \{SIs\} := \{SIs\} ∪ ReadPrev (d, version)
}
ReadPrev (d, version) {
    repeat {
        foreach s in the k+2f nearest nodes of $R_d$
            send READPREV message to $s$
    } until (collect k+2 responses)
    return \{SIs\} in responses
}
Node.processMessage (message) {
    else if (message.type = READLAST)
        Respond last SI of $d$
    else if (message.type = READPREV)
        Respond max SI lower than version
}
```

Figure 2. Read_OVN algorithm.
380 pairs of nodes that are distributed globally. The experimental results are shown in Figure 3.

![Distance and Ping Latency](image)

**Figure 3. Communication cost measurement.**

The results are analyzed and a fitting function is derived to describe the relationships:

\[ F_1: \text{PingLatency} = 0.4 + 0.3 \times (\text{Distance})^{0.735} \]

Then we explored the relationships between the total communication latency and the ping latency and message size. We set up socket communication among 5 sites all over the world, a total of 16 communicating pairs, and measure the latency for different message sizes. Based on the results, a fitting function is derived as follows:

\[ F_2: \text{Latency} = 0.02 \times (\text{Size})^{0.53} \times \text{PingLatency} + \text{PingLatency} \]

These two functions are used to simulate the transmission latencies for given geographical distances and the message sizes.

### 6.2 Experimental Results

Based on the simulation model discussed above, we conduct a series of experiments to study and compare the system performance for the three access protocols. The metrics for the comparisons are read access latency and update access latency.

#### 6.2.1 Access Latency

In all the experiments, the Read_COVN, Read_DOVN, and Read_DUVN algorithms have almost the same performance because they all require the same number of messages and one round of message exchanges in optimistic situations. The update latency for Update_DUVN is always the lowest since Update_DUVN does not have the cost of reading the version numbers. Update_COVN normally has the highest latency due to the drawbacks of centralized approach.

We first study the performance of the algorithms under different \( k \) settings. The performance results for the three read access algorithms and three write access algorithms are presented in Figures 4 and 5, respectively. In a parallel access, the access latency is dominated by the message with the highest latency. In Figure 4, the read latency increases as \( k \) increases because the client nodes had to request more data shares from farther nodes although the communication overhead decreases as \( k \) increases. But in Figure 5, the update latencies for Update_DUVN, Update_COVN and Update_DOVN show different trends. The update latency for Update_DOVN is determined by latencies for the version number request and the write share messages. When \( k \) is small, the latency decreasing of write share messages dominates the latency increasing of version number request messages since the write share message size decreases rapidly. So, at first, the update latency for Update_DOVN decreases as \( k \) increases. But when \( k \) becomes large, the trend is reversed. The latency increasing of version number requests become dominating, so the total latency increases. But for Update_COVN, the communication latency to the version server is fixed and not affected by \( k \). And the update latency for Update_DUVN only includes the write share message latency, so both of them keep decreasing as \( k \) increases.

![k and Read Latency](image)

**Figure 4. Relationship between \( k \) and read access latency \((n=29)\)**

![k and Update Latency](image)

**Figure 5. Relationship between \( k \) and update access latency \((n=29)\)**

Next, we study the impact of storage node failures on the performance of various access protocols. The results for the three read and update algorithms are presented in Figures 6 and 7, respectively. When the node failure rate \( f \) increases, more messages are required for each read request and for each version number request, so the read request latency and update latency for Update_DUVN increases. However, as can be seen in the figure, the node failure rate does not affect the latency for Update_COVN and Update_DUVN.
Finally, we study the impact of the update access rate on the performance of the access algorithms. The performance results for the three read and the three update algorithms are presented in Figures 8 and 9, respectively. The update access rate does not change the required ping latency and message size for one access, so the read and update latencies do not have obvious changes.

7 Conclusion
In this paper, we consider integrating SSS and DHT technologies for P2P storage systems design to meet the data availability and security requirements and provide efficient data accesses. We design a DHT algorithm for share location and search. Due to the problems with partial share updates and concurrent update, we consider three access protocols based on different version number models and conduct experimental studies to compare their performance.

The performance results for the read accesses of the three protocols are very similar. But for update access latency, the DUVN protocol always has the best performance. But as discussed earlier, DUVN may incur a higher storage cost for storing more SIs.

8 References


[Globus] [www.globus.org](http://www.globus.org)


[Inet3] [http://topology.eecs.umich.edu/inet/](http://topology.eecs.umich.edu/inet/)


[Ns2] [http://www.isi.edu/nsnam/ns/](http://www.isi.edu/nsnam/ns/)


