Deleting Files in the Celeste Peer-to-Peer Storage System

Gal Badishi[*†] Germano Caronni[‡] Idit Keidar[†] Raphael Rom[†‡] Glenn Scott[‡]

[†]*Department of Electrical Engineering, The Technion – Israel Institute of Technology.*

[‡]*Sun Microsystems Laboratories.*

Contact Author:
Gal Badishi
Department of Electrical Engineering
Technion – Israel Institute of Technology
Haifa 32000, Israel.
badishi@ee.technion.ac.il
Phone: +972-4-8293298
Fax: +972-4-8295757

[*]Supported by the Israeli Ministry of Science.

Abstract

Celeste is a robust peer-to-peer object store built on top of a distributed hash table (DHT). Celeste is a working system, developed by Sun Microsystems Laboratories. During the development of Celeste, we faced the challenge of complete object deletion, and moreover, of deleting "files" composed of several different objects. This important problem is not solved by merely deleting meta-data, as there are scenarios in which all file contents must be deleted, e.g., due to a court order. Complete file deletion in a realistic peer-to-peer storage system has not been previously dealt with due to the intricacy of the problem – the system may experience high churn rates, nodes may crash or have intermittent connectivity, and the overlay network may become partitioned at times. We present an algorithm that eventually deletes all file content, data and meta-data, in the aforementioned complex scenarios. The algorithm is fully functional and has been successfully integrated into Celeste.

Keywords: peer-to-peer, storage, fault-tolerance.

1 Introduction

Two different technologies have been developed in recent years: network storage systems and peer-to-peer networking. Network storage, exemplified by a variety of NAS and SAN products, is a result of the realization that stored data is sufficiently valuable that reliable and continuous access to it is mandatory. Peer-to-peer systems have evolved to create large distributed systems from small and unreliable components, overcoming the cost of providing extremely reliable units. It was only a matter of time before these technologies merge to create peer-to-peer based storage systems. Examples of such systems, each emphasizing a different aspect of data storage, are Farsite [1], OceanStore [4], Venti [8], Freenet [3], and Ivy [5].

Each of these systems contains an overlay network, typically constructed on top of a distributed hash table (DHT), providing routing and object location services to a storage management system. Most of these storage systems consider aspects of data replication, reliability, security, and storage maintenance, but almost none of them addresses data deletion directly. It is noteworthy that Plaxton et al. [7], in their seminal paper, do address data deletion at the DHT level. However, their system is static, rendering deletion a much easier problem than in a dynamic system as considered herein. Additionally, their system does not address more complex (and perhaps higher level) issues of security, trust, and access control, which we consider important. We provide some ideas on how to secure the deletion process in real systems in Section 6.

One can identify three tiers of data deletion. The first tier is access-based deletion, in which data is not actually removed but access to it is made harder. File systems typically delete pointers to the data (sometimes by modifying the file's meta-data). Another approach is to use data encryption in which case data deletion amounts to destroying the relevant keys [6]. This is the easiest of all tiers and relies on the inability to ever access data without pointers to it or decrypt data without knowing the relevant keys. In some cases, access-based deletion may be insufficient, such as due to court orders. Company lawyers, such as Sun Microsystem's, often demand that a storage system will have the ability to completely delete the contents of a file, so as to comply with the judge's ruling.

The second tier of deletion is data obliteration, in which data itself, not just the access means to it, are completely removed from the system. It is a better approach to data deletion as it is independent of future technological advances. This tier does not necessarily replace the first tier, but rather augments it with much stronger deletion guarantees.

The third tier is data annihilation, in which all traces of the data are removed from all media on which it is stored. This tier is extremely costly to implement and is typically reserved to national security related data. In this paper, we deal with data obliteration, i.e., the second tier.

Robustly obliterating a file in a survivable peer-to-peer storage system is a real challenge. The main difficulty lies in the mere fact that the storage system is designed to be survivable and to provide availability even when nodes crash and links fail. To allow that, the system usually cuts a file into multiple objects, replicates all of them, and stores their copies on different nodes. To enable complete deletion, all of these chunks must be located and deleted. What is worse is that the storage system might try to maintain a minimum number of copies available, so as to guarantee availability. This stands in contrast to what the deletion algorithm wishes to do. Additionally, nodes may join or leave the system at arbitrary times, in an orderly fashion, or simply by crashing or becoming part of a distinct network partition. The objects these nodes hold might re-enter the system at unknown times in the future. By this time, the node at which the deletion request was initiated may be unavailable. The system should always know whether an object that has entered the system should in fact be deleted. Finally, secure deletion means that only authorized nodes should be allowed to delete a specific file.

We present a deletion (in the sense of data obliteration) algorithm designed to operate on top of typical DHTs. The algorithm was implemented in Celeste [2], which is a large scale, distributed, secure, mutable peer-to-peer storage system that does not assume continuous connectivity among its elements. Celeste was designed and implemented at Sun Microsystems Labs. Our deletion algorithm is founded on a separate authorization and authentication mechanism to allow deletion of stored objects. It is based on secure deletion tokens, one per object, that are necessary and sufficient to delete an object.

In Section 2, we describe the system in which our deletion algorithm operates, namely, a storage system, (Celeste in our case), running atop a DHT, as well as the cryptographic authorization and authentication mechanisms used. Section 3 presents our deletion algorithm. This section describes the algorithm abstractly, without linking it to a particular implementation.

In Section 4, we formally prove sufficient conditions for the algorithm's success. Roughly speaking, we prove that in a stable network partition P that includes at least one copy of each of the meta-data objects describing a file F, if any node in P tries to delete F (at any point in time), then the algorithm guarantees

complete deletion of the contents of all data and meta-data objects associated with F in P. Moreover, when such a partition P merges with another partition P', our algorithm ensures complete deletion of all data objects associated with F in $P \cup P'$.

In Section 5, we validate the effectiveness of the algorithm through simulations in typical settings. First, we present static simulations showing that in a stable partition, complete deletion is achieved promptly. These simulations validate the correctness proof of Section 4, and demonstrate the deletion time in stable situations, where eventual complete deletion is ensured. Moreover, the algorithm is scalable: as the underlying communication is based on a DHT overlay, deletion time increases like the routing time over a DHT – logarithmically with the number of nodes. We further show that the deletion time increases linearly with the number of versions the file has, but decreases with the number of replicas, since each node holding a copy of a meta-data object of a file participates in the deletion process. At the same time, a larger number of replicas increases the message load on the system. Secondly, we simulate the algorithm in dynamic settings, where nodes constantly crash and recover. These simulations enact scenarios that are not covered by the correctness proof of Section 4, which only considers eventually-stable settings. Nevertheless, in these simulations, complete file deletion is always successful. Moreover, the time it takes for a file to be completely obliterated from the system is proportional to the time it takes failed nodes holding the file's objects to recover.

Finally, Section 6 addresses some implementation issues that arise when implementing our algorithm in a real system, namely Celeste. Section 7 concludes.

2 System Architecture

We model our peer-to-peer system as a dynamic, intermittently connected, overlay network of nodes that may join or leave the system in an orderly fashion and can also crash unexpectedly. Additionally, the overlay network may become partitioned due to link failures. Crashed nodes may come back to life, retaining their previous state. In this paper we do not deal with nodes that transfer state information from other nodes. Such a state transfer will only facilitate the deletion process, and thus our results provide a lower bound on the algorithm's robustness.

Our system is composed of two layers, similar to Oceanstore [4, 10]. The lower layer is a DHT that is used for locating, storing and retrieving objects. Any DHT can be used, as long as it provides the interface

and semantics described below. Exemplar DHTs are Tapestry [13], Pastry [11], Chord [12], and CAN [9]. Above the DHT layer resides the *Celeste* layer [2]. Celeste provides a secure, survivable, mutable object store utilizing the DHT.

The Celeste layer. Each object in the DHT has a *global universal identifier* (GUID), e.g., a 256-bit number. A Celeste object is comprised of two parts, the data and the meta-data. The meta-data contains information about the object, and is used both by the DHT and by Celeste. (Note that Celeste's meta-data is different from the filesystem's notion of a file meta-data, which is typically stored in a separate Celeste object). The integrity of objects in the DHT can be verified, and nodes do not store objects that fail an integrity check. With respect to verifiability, an object belongs to one of two categories: (1) A *self-verifiable object* is an object whose GUID is a hash of a mixture of its data section and its meta-data section. (2) A *signed object* is an object whose GUID is arbitrary, but its meta-data contains a digital signature (issued by the object's creator/modifier) that allows verification of the object's integrity (see Section 6). It is important to note that no two objects have the same GUID. Not even self-verifiable objects with an identical data section (their meta-data section is different).

General information about the file is saved as an object of a special type, called an AObject. Among others it includes the GUID of the latest version. Thus the GUID of the AObject (AGUID) is a lead to all the versions of the file. Each file update generates a new version containing information on that update. Each version is described by a special object called a VObject, whose GUID is called a VGUID. The AObject contains the VGUID of the latest version of the file, and each VObject contains the VGUID of the previous version of the file. Since the AObject is mutable but maintains the same GUID, it is a signed object. In contrast, VObjects are self-verifiable. We note here that having versions is very similar to having different files represent different updates to the same core file. In that respect, versioning is just a property of our system, and can be replaced by other means for file updates.

For simpler storage and management, the file's data contents are divided into blocks. Each block is stored as a separate self-verifiable object called a BObject, whose GUID is called a BGUID. Each VObject contains a list of all the BGUIDs that hold the data for that version of the file. The relations among these entities are depicted in Figure 1. Note that all Celeste objects are replicated in the DHT.

The DHT layer. The DHT layer provides the Celeste layer with primitives of object storage and re-

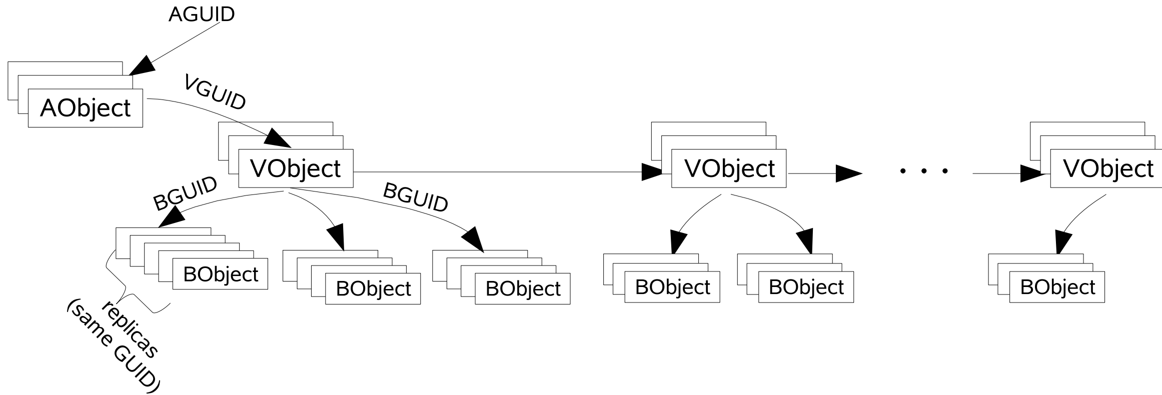


Figure 1: The relations between Celeste objects comprising a file.

trieval. The DHT recognizes nodes and stored objects. Every object is assigned a *root* which is uniquely chosen among the live nodes in the system. The root is determined based on the object's GUID. (Nodes and objects have GUIDs drawn from the same space.) For example, the object's root can be the live node with the lowest GUID that is not lower than the object's GUID (if no such node exists, then the root is the node with the lowest GUID).

The root of an object is ultimately responsible to track the whereabouts of copies of the object. To that end the root of an object keeps soft-state information about the location of each copy of the object in the system, through the use of *backpointers* – a list of nodes that hold the object's copies (for efficiency, these backpointers may also be cached on other nodes).

A node's DHT layer provides its Celeste with the ability to send messages to Celeste layers on other reachable nodes, as well as with the following local functions:

- *storeObject*. Stores an object locally and sends a *publish* message to the object's root. The publish happens both immediately and periodically, as long as the object is stored. The publish message allows the root to maintain its soft-state tracking information, by saving a backpointer to the publisher. Backpointers expire after some time, to make sure only up-to-date information is saved. Thus, it is important for the publish rate to be sufficiently high to avoid unnecessary backpointer expiration in the root.

- *retrieveObject*. Utilizes a backpointer in the object's root or on the way to it, to fetch a copy of the

object if one exists.

Problem definition. We tackle the problem of completely deleting file contents from a peer-to-peer storage system. The deletion entails removing all the versions of the file, their contents and their associated meta-data, limiting the operation to authorized entities. The challenge lies in securely obliterating all these data in a survivable, dynamic, failure-prone system that continuously strives to keep replicated objects available even when nodes crash and links fail.

Mechanisms for deletion. Our object deletion mechanism relies on a *deletion token* (DT) and its hashed version–the *deletion token hash* (DTH). A deletion token may be any number, e.g., a random 256-bit number. A single deletion token is associated with every object in Celeste (but different objects may have the same deletion token). The object's deletion-token hash, i.e., the hash of the deletion token, is stored in the object's meta-data. Conversely, the object's deletion token is kept away from the object prior to deletion. Celeste considers a user authorized to delete an object if it presents the object's deletion token. A correct node agrees to delete an object only when supplied with a deletion token that hashes to the object's deletion-token hash stored in the object's meta-data. For efficiency reasons, in Celeste all objects belonging to the same file have the same deletion-token. To provide for secure deletion, the file's deletion token is only known to entities that are allowed to delete the file. The aforementioned mechanisms for object integrity verification must include the deletion token hash.

A user that wishes to delete an object needs to expose the object's deletion token by supplying it to nodes that hold the object. Once a deletion token is exposed, it is no longer secret, and every node can use it to request other nodes to delete copies of the object (obviously containing the corresponding deletion-token hash). An exposed deletion token in an object's meta-data means that the deletion procedure for that object is underway, but it may take some time until all copies of the object are actually deleted, due to the distributed nature of the system.

3 Deletion Algorithm

This section discusses the algorithm for deleting a file. The algorithm is presented in pseudocode in Figures 2 and 3. Implementation issues are deferred to Section 6.

```
deleteFile(AGUID a, exposed token t):
   create deletion object O with
         GUID=a, deletionToken=t, type=AObject
   deleteObject(O)

deleteObject(object D_O):
   if !isValid(D_O) then return
   /* if object is already deleted do nothing */
   if isStoredLocally(D_O.GUID) then
         O = getLocalObject(D_O.GUID)
         if O.deletionToken! =null then return
   /* handle object deletion according to type */
   if D_O.type == AObject then
         deleteAObject(D_O.GUID, D_O.deletionToken)
   else if D_O.type == VObject then
         deleteVObject(D_O.GUID, D_O.deletionToken)
   else
         storeObject(D_O)

/* Deletion Procedure at the Object's Root */
Upon receiving publish message for object O from N:
   if O.deletionToken! =null then
         if isDeleted(O.GUID) then return
         if !isValid(O) then return
         deleteObject(O)
         for each backpointer! =me in backpointers(O.GUID) do
               send deletion request with O to backpointer
         markDeleted(O.GUID)
   else /* O is not deleted */
         if isDeleted(O.GUID) then
               D_O = getFromLocalStore(O.GUID)
               send deletion request for D_O to N

   Upon receiving deletion request with D_O:
         if !isStoredLocally(D_O.GUID) then return
         deleteObject(D_O)
```

Figure 2: Basic deletion procedures at each node.

The deletion of a file starts at one or more (authorized) nodes and propagates to all the nodes that hold the file's objects throughout the system. To start the deletion sequence, a node invokes *deleteFile* with two parameters: the file's AGUID, and an appropriate exposed deletion token. To get the gears of the deletion process in motion, all that *deleteFile* does is create one copy of an AObject corresponding to that file with an exposed deletion token. Such a copy is called a *deletion object*. A deletion object with GUID g serves as a permanent indication that the object whose GUID is g must be deleted. Creating such a deletion object does not necessitate fetching a copy of the original object; rather, an empty object with the same GUID and type and an exposed deletion token is created. After creating such an object, *deleteFile* invokes the procedure *deleteObject* for it.

Let us now overview the steps taken in the deletion of *any* object (cf. *deleteObject*, Figure 2). A node that begins the deletion process for an object O is called a *deleter of O*. The deleter must have O's valid deletion token. If the deleter has already deleted O in the past, (in which case it has a local copy of O with an exposed deletion token), then it performs no further operations. As explained earlier, an object O is deleted by storing a deletion object with O's GUID and an exposed deletion token. When an AObject or VObject is deleted, the deleter does some extra processing before storing this object, in order to ensure that the objects (VObjects and BObjects) that O points to will also be deleted, as will be explained shortly.

After creating a deletion object D_O, the deleter calls the DHT's *storeObject* procedure in order to store a local copy of the deleted object. If a local copy of O already exists, then *storeObject* replaces it by D_O, since O and D_O have the same GUID. In addition, *storeObject* issues a *publish* message, which winds its way to O's root, (again, since O and D_O have the same GUID, they have the same root). The deletion process is then continued by the root.

Thus, the mechanism of locally storing an object with an exposed deletion token serves as a persistent way to notify the object's root that the object should be deleted. Note that in a fault-prone dynamic system, an object's root may fail (crash) before completing the deletion, or even before being notified. But in this case, another node becomes the new root and since every Celeste replica periodically broadcasts a publish message toward the object's current root, the new root is eventually notified, and the deletion continues.

The root begins to engage in the deletion process *upon receipt of a publish message* for an object O with an exposed deletion token. If the root has already deleted the object with O.GUID in the past, or if O does not have a valid deletion token, then the root performs no further operations. The root deletes the object by calling the *deleteObject* procedure discussed above. The root then sends a *deletion request* message to each of the other nodes holding O, according to the known backpointers for O. The deletion request contains the published information for O, including the exposed deletion token and GUID. Finally, the root marks the deletion, that is, stores the information that O.GUID has been deleted. If the root subsequently receives a publish message for O, then it sends a deletion request to the node that sent the publish message.

Upon receiving a deletion request for O, the receiving node checks that it has O. If so, it deletes it using the *deleteObject* procedure.

AObjects are deleted in the procedure *deleteAObject* (see Figure 3). As before, the object is deleted by creating a deletion object with the object's GUID and an exposed deletion token, and storing it using the DHT's *storeObject* procedure. However, if the deleter holds a local copy of the object, then it cannot simply over-write it with a deletion object. This is because the AObject is the head of the linked list that contains all the file's VGUIDs, i.e., it is the head of the list that points to each version of the file. Thus, if the deleter has a local copy of the deleted AObject, it first initiates the deletion process for O.lastVGUID (by calling *deleteVObject*), and only then deletes O's contents and stores it locally. Several "last" versions may exist due to partition merging, all corresponding to the same single AObject (see [2] for more details).

```
deleteAObject(AGUID a, deletion token t):
  /* If local copy exists, delete last VObject(s) */
  if isStoredLocally(a) then
      O = getLocalObject(a)
      for each lastVGUID in O.lastVGUIDs do
          deleteVObject(lastVGUID, t)
  /* Store deletion object */
  create deletion object O with
      GUID=a, type=AObject, deletionToken=t
  storeObject(O)
```

```
deleteVObject(VGUID v, deletion token t):
  if isStoredLocally(v) then
      O = getLocalObject(a)
      for each b in O.BGUIDs do
          create deletion object BO with
              GUID=b, deletionToken=t, type=BObject
          deleteObject(BO)
      for each prevVGUID in O.prevVGUIDs do
          if prevVGUID != null then
              deleteVObject(prevVGUID, t)
  /* Store deletion object */
  create deletion object O with
      GUID=v, type=VObject, deletionToken=t
  storeObject(O)
```

Figure 3: Deletion procedures for AObjects and VObjects.

VObjects are deleted in the procedure *deleteVObject* (see Figure 3). As before, the deleter creates a deletion object with the object's GUID and an exposed deletion token, and stores it. As with AObjects, before a node N deletes a VObject V_i from its local store, it must ensure that earlier versions will eventually also be deleted. First, N deletes all the BObjects that are part of version i, by creating corresponding deletion objects and calling *deleteObject* for each of them. Then, N checks whether an earlier version, V_{i-1} was indeed created (several such versions may exist, due to merging). If so, it continues with the deletion process by calling *deleteVObject* for V_i. Finally, N deletes the contents of the object and stores it with an exposed deletion token.

4 Correctness

In this section, we discuss the deletion algorithm's guarantees: we show *sufficient* conditions for complete deletion of a file. Although, as the simulations in the next section show, the algorithm succeeds in deleting files even in highly dynamic settings, it is very hard to reason about the algorithm in such settings. For the sake of reasoning about the algorithm, we examine *stable* periods. That is, we examine runs in which the system eventually stabilizes (the stabilization can occur either before or after the deletion initiation). Formally, stabilization is defined as follows:

Definition 1 (Stability)**.** *We say that partition P (consisting of a set of nodes) is stable in a time interval if in that interval all nodes can communicate with each other, nodes do not join or leave P, and no versions*

are added to files. We say that partition P is eventually stable from time T onwards if it is stable in $[T, \infty)$.

Note that in particular, nodes do not crash and crashed nodes do not recover in P in a stable interval. We do not explicitly state the interval when it is clear from context.

In a partition P, our requirement from the algorithm is to completely delete the contents of all replicas of all objects pertaining to a deleted file. Formally:

Definition 2 (Deletion)**.** *An object copy is said to be deleted if its deletion token is exposed in its meta-data section and it has empty data contents. An object is deleted in partition P if all its copies in P are deleted. A file F is deleted in P if all objects pertaining to F are deleted in P.*

The dynamic failure-prone nature of the system renders the deletion problem (in the obliteration sense) unsolvable in some scenarios. Consider, for example, a file with two versions, and a stable partition P in which no node holds a copy of last version's VObject. Since the only pointer to the first version is in that VObject, there is no way for the algorithm to discover the first version's objects, even if they do exist in P, and hence, there is no way to delete them. Therefore, in order to identify partitions where complete deletion is possible, we examine the nodes to which objects are *assigned* by Celeste when they are first stored, or at any time prior to the initiation of a deletion algorithm that should erase them:

Definition 3 (Assigned copies)**.** *A node N is an assigned copy of object O pertaining to a file F if a copy of O is stored at N at a time t so that* deleteFile(F.GUID) *is not invoked before time t.*

This definition allows us to define a sufficient prerequisite for complete deletion of a file F in a partition P:

Definition 4 (Complete-deletion prerequisite for file F)**.** *We say that partition P fulfills the complete-deletion prerequisite for file F, if F's AObject is assigned to some live node N in P, N has received the up-to-date version of F's AObject (pointing to the most recent version), and for each version that belongs to file F there is a live node in P to which the corresponding VObject is assigned.*

We now turn to state the algorithm's guarantees. The proofs of the following lemmas and theorem are given in Appendix A.

Lemma 1. *Let P be a partition that is eventually stable from time T, and let O be some object in the system. If after time T a live node in P has the deletion object for O, D_O, then all nodes holding (undeleted) copies of O in P eventually receive a deletion request for O.*

Corollary 1. *Let P be a partition that is eventually stable from time T, and let O be some object in P. If after time T a live node in P has the deletion object for O, then all copies of O in P are eventually deleted.*

Lemma 2. *Let V be some version of file F, and let $VObj$ be the VObject representing this version. If the following holds for partition P: (1) P is eventually stable from time T; (2) P fulfills the complete-deletion prerequisite for file F at time T; and (3) A live node in P holds a copy of the deleted $VObj$, D_{VObj}, at time T, then all objects in P associated with version V and all earlier versions of F are eventually deleted.*

Theorem 1. *Let F be a file, and let $AObj$ be the AObject representing F. If the following holds for partition P: (1) P is eventually stable from time T; (2) P fulfills the complete-deletion prerequisite for file F at time T; and (3) A live node in P holds a copy of the deleted $AObj$, D_{AObj}, at time T, then all objects in P associated with file F are eventually deleted.*

We get the following Corollary:

Corollary 2. *Let F be a file, and let $AObj$ be the AObject representing F. If the following holds for partition P: (1) P is eventually stable from time T; and (2) P fulfills the complete-deletion prerequisites for F at time T, then if a node N in partition P tries to delete file F at* any *time τ (before or after T), all objects in P associated with file F are eventually deleted.*

Finally, we get that deletion is ensured not only in one stable partition, but also when partitions merge:

Corollary 3. *Let F be a file, and let P_1 and P_2 be two partitions that merge into a single partition P at time τ, such that: (1) P_1 is stable in the interval $[T, \tau]$; (2) P_1 fulfills the complete-deletion prerequisite at time T; and (3) P is eventually stable from time τ. If a live node in P_1 holds a copy of the deleted $AObj$, D_{AObj}, at time T, then all objects in P associated with file F are eventually deleted.*

5 Simulation Results

Our deletion algorithm is implemented in Celeste, and was successfully tested in an environment containing a relatively small number of nodes. However, we would like to analyze the algorithm's expected behavior in actual deployment scenarios where many nodes are present. To test the effectiveness of our algorithm, we simulate it in diverse settings. The simulation is round-based. In each round, nodes perform whatever

processing they need to do, and messages propagate at most one hop in the overlay network. Since the system is intended to be deployed in WANs, a round represents about 200 msecs. In each experiment, nodes publish their locally stored object copies every 20 seconds (100 rounds), and backpointers expire 100 seconds (500 rounds) after receiving the last publish message. Each data point in each graph represents the average result of 500 experiments.

The system is composed of a single partition. In some experiments the partition is static, while in others nodes leave and join the partition by crashing and coming back to life, respectively. Messages sent between nodes are routed through intermediate nodes in the overlay. The number of hops it takes a message to reach its final destination is drawn from a bell-shaped distribution with a minimum of a single hop and a maximum logarithmic in the number of nodes. This latency distribution is typical in peer-to-peer overlays, and resembles one measured in Chord [12].

At the beginning of each experiment, a file is stored on random nodes. That is, all copies of each of the objects representing the file are stored on randomly-chosen nodes in the partition, such that no two copies of the same object are stored on the same node. After storing the file, an initial state, i.e., running or crashed, is selected for each node, according to the scenario that is being evaluated. After the nodes run for a while, a live node chosen uniformly at random attempts to delete the file.

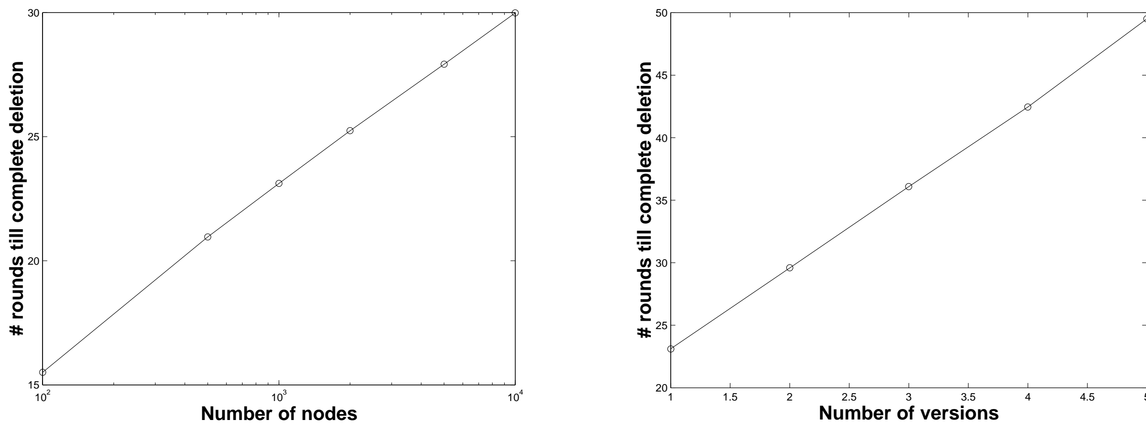
We measure the *time till complete deletion*, i.e., the amount of time from the moment that some node initiates the deletion sequence, until all copies of objects associated with the file are deleted.

We do not consider nodes joining and leaving the partition, but rather nodes crashing and either staying down or coming back to life with their previous state intact. A node that comes back to life with its previous state, immediately publishes all of its objects and expires all of its stale backpointers. We note that this is a worst-case approach, which gives us an upper bound on the time till complete deletion. Obviously, the deletion time would not be higher had recovered nodes received state updates upon recovery.

We first evaluate our protocol in a static partition where all nodes remain up. This allows us to validate our simulation and provide insight on the impact of several parameters on the algorithm's performance. We then turn to examine our protocol in a dynamic setting, where nodes constantly crash and recover.

Failure-free operation. We first evaluate our deletion algorithm in a static, failure-free partition. Unless otherwise noted, the replication count is 5, and the file is represented by 1 AObject, 1 VObject, and 10

BObjects. I.e., the file has a single version with 10 data blocks, and there are 60 object copies stored in the DHT and associated with the file.



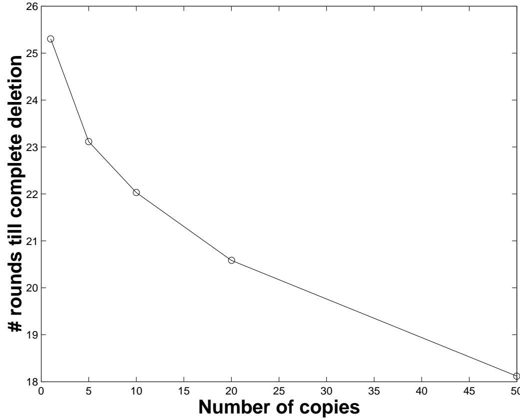
(a) Dependency on number of nodes (log), 1 version. (b) Dependency on number of versions, 1000 nodes.

Figure 4: Average time till complete deletion, 10 blocks per version, 5 copies per object.

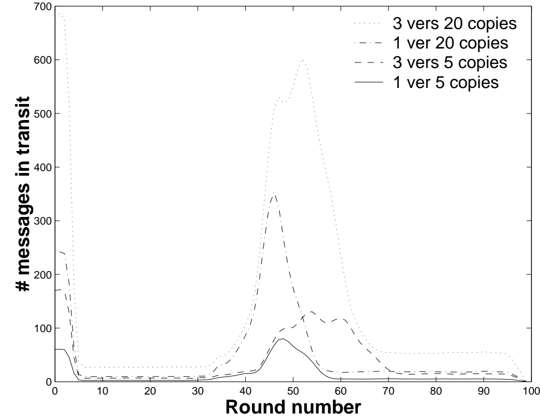
Figure 4(a) shows the time in rounds till complete deletion as a function of the number of nodes. Since the number of nodes is depicted on a logarithmic scale, we can see that the deletion time increases linearly. Due to this scalability, there is little importance in choosing a specific number of nodes to simulate. Henceforth, we simulate 1000 nodes. Figure 4(b) shows the time till complete deletion as a function of the number of versions. As expected, each additional version incurs the same overhead. Henceforth, we evaluate the algorithm for files with a single version. We have also measured the impact of increasing the number of of blocks (BObjects) per version, and found that it has a negligible effect on the time till complete deletion. (The graph is not shown here due to space considerations).

Figure 5(a) depicts the time in rounds till complete deletion as a function of the number of replicas per object. We can see that the more replicas there are, the faster the deletion completes. It is obvious that the deletion time should not increase as the number of copies increases, as all object copies are deleted roughly in parallel by simultaneous deletion requests sent from the object's root. The decrease in deletion time happens since having more copies per object increases the chance that some node holds copies of both the AObject and the VObject. Since the file has only a single version, all object copies associated with the file can be deleted in parallel when that node calls $deleteAObject$.

Figure 5(b) shows the number of messages in transit, i.e., the number of messages that have not yet



(a) Dependency on number of replicas, 1 version.



(b) Message overhead.

Figure 5: Average time till complete deletion and message overhead for 1000 nodes, 10 blocks per version.

reached their final destination. The peak at round 0 are all the publish messages sent by the nodes storing the object copies for the file. The number of publish messages is exactly the number of objects across nodes. For example, for a single version with replication factor of 5 we get 60 stored object copies. This is an artifact of our experiments, where nodes "wake up" at time 0 having an object, and immediately publish it. The number of messages quickly drops and stays low, as only periodic publish messages are sent. These periodic messages are uniformly distributed among all particvipating node, over the 20-round publish period. The deletion is initiated at round 30 by a node chosen uniformly at random.

We see that the more versions we have, the longer it takes the system to become quiescent. This is due to the mostly-sequential nature of deleting several versions. We also see that when there are more replicas, more messages are sent per round. This expected behavior occurs since all backpointers are notified simultaneously, and all nodes storing a deleted object immediately publish it. The number of deleted objects is higher than the number of object copies, since nodes, e.g., roots, may hold deleted objects simply as an indication that the object is scheduled for deletion. This is the reason for the increase in the number of messages when the deletion completes, compared to the number of messages before the deletion starts – there are simply more objects to publish. Finally, the number of messages drops to 0 when the round number approaches 100. This is simply an artifact of the experiments being shut down.

The results of this section validate Theorem 1. We further note that when the assumptions of Theorem 1 hold in a partition, the time till complete deletion after the partition stabilizes (i.e., from time T onwards) is

no worse, and often better than the time till complete deletion measured in our failure-free evaluations. This is due to the smaller number of live nodes in the partition.

Dynamic simulations. We now simulate the algorithm in a partition that is never eventually stable. Each node is either crashed or running at any given time, and crashed nodes eventually run again, and vice versa. Node session times (time to failure) and down times (time to recovery) are distributed exponentially. When crashed nodes recover, they hold the same objects and backpointers they had when they crashed, and they immediately publish their stored objects and expire stale backpointers. We measure the time till complete deletion, that is, we must wait for crashed nodes holding parts of the file to recover before we can declare success. In all of our simulations, all of the files were eventually completely deleted. Thus, our algorithm achieves its goals under broader circumstances than formally established in the previous section.

We start all experiments in the partition's steady state, i.e., the node's initial state, up or down, is determined with respect to its uptime - the percentage of time in which the node is alive. Figure 6 shows the average time till complete deletion as a function of the nodes' uptime, where all nodes have the same uptime. Each node's average session time is 1 hour, i.e., a node fails on average after one hour of operation. We observe that the higher the uptime, the faster the deletion process completes. This is expected, as there is a lower probability of a data object residing on a crashed node at deletion time. When an object resides on a crashed node, the deletion process cannot complete before that node recovers. Having understood the dependence on uptime, we fix it at 99% for our next experiments, and measure the effect of other parameters.

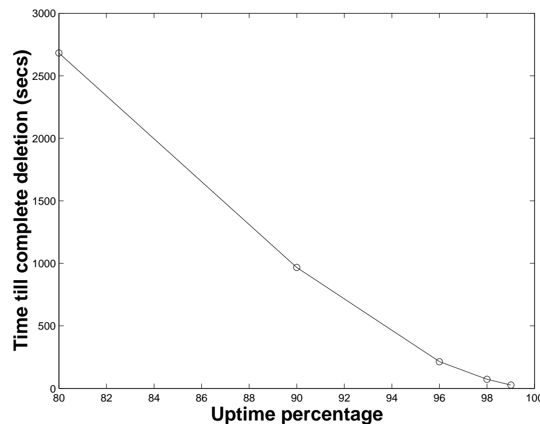
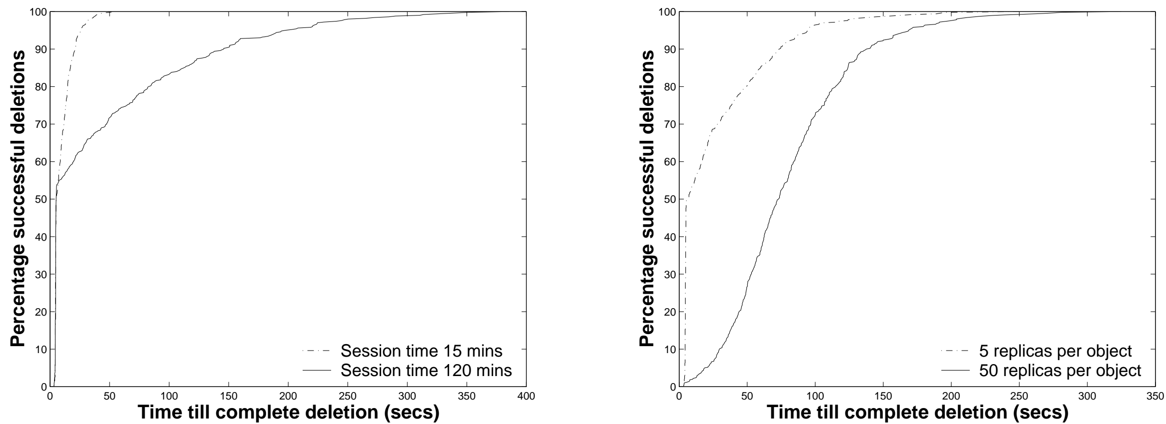


Figure 6: Average time till complete deletion vs. node uptime, 1000 nodes, 1 version, 10 blocks per version, 5 copies per object, average session time 1 hour.

Figure 7 shows CDFs of the time till complete deletion measured for 500 deletions. Figure 7(a) shows the dependency on the nodes' average session time. The longer the session time, the longer it takes to delete the file. Since the average uptime is fixed to 99%, a longer session time means that if a node is down when the deletion request is issued, it will take the node longer to come back to life. This is exactly the time in which the data object is unreachable and cannot be deleted. Once the node comes back to life, the object it holds can be deleted promptly.

Figure 7(b) shows the how the number of object copies affects the time till complete deletion in a failure-prone system. We can see that fewer replicas mean better deletion times, in contrast to the result we get in a failure-free system, as depicted in Figure 5(a). The reason for this is that having more copies in a failure-prone system increases the probability that a crashed node has an object that needs to be deleted. The deletion process only completes when all such crashed nodes come back to life and receive deletion requests for their objects.



(a) Dependency on session time, 5 copies per object.(b) Dependency on replication, 1 hour average session.

Figure 7: CDFs of 500 deletions showing dependency on session time and replication, 1000 nodes, 1 version, 10 blocks per version, 99% node uptime.

6 Implementation Issues

We have implemented the deletion algorithm presented in Section 3 in Celeste, a large-scale, peer-to-peer, secure mutable storage system developed in Sun Microsystems Labs. We now present some implementation specifics and discuss their impact on the deletion algorithm. Celeste is implemented in Java. It uses the hash

function SHA-256 to generate GUIDs for nodes and objects.

Erasure coding. In addition to replication, Celeste supports erasure coding as a way of providing redundancy with reduced storage costs. In this case, a VObject points to *fragments* in addition to blocks. The fragments are used when the blocks are unavailable (unreachable). Fragment objects (FObjects) are treated by the deletion algorithm the same way as BObjects are. When an object is fragmented, the GUIDs of its fragments are stored alongside the original GUID itself (e.g., in the AObject or in another VObject). If the VObject fragments are redundant, i.e., a complete VObject exists as well, then they are treated like FObjects. Otherwise, before deleting an object that points to a fragmented VObject, the fragmented VObject is reconstructed (by retrieving sufficiently many fragments), stored locally, and deleted using *deleteVObject*. Inability to reconstruct the VObject is equivalent to it being unreachable. In such cases, Celeste relies on application users to re-try deletion. In general, erasure coding makes the deletion process easier, since each fragment has a different GUID and is not replicated.

Refreshing mechanism. Since some nodes may permanently depart from the system, Celeste implements a *refreshing mechanism* to ensure that enough copies (or fragments) of a given object are available. To this end, it supports the role of a *refresher*, a node that creates new replicas when their number in the partition is too low. In order to work well with the deletion algorithm, we dictate that the refresher be the object's root. Thus, once the root is notified of a deletion it ceases to refresh the object.

Access rights. Celeste provides support for version access rights using encryption and digital signatures. For each version of a file, Celeste maintains a list of nodes that may either read, modify, or delete the version. When embarking on the deletion of a file, Celeste first deletes the list data used to access the file, effectively causing the file to be inaccessible. (Note that this is what we call tier one deletion).

Preventing over-writing of deleted objects. Recall that our deletion algorithm does not directly delete objects, but rather replaces them with empty objects that have the same GUID and the corresponding exposed deletion token in their meta-data. Once this has happened, it is important to ensure that the the deletion object will not be over-written by another (new) data object having the same GUID, since the correctness of our algorithm depends on the existence of such a deletion object. To this end, we have modified the DHT's *storeObject* function so that it does not allow over-writing an object with an exposed deletion token.

Protection against malicious attempts to prevent deletion. Although our algorithm is not intended for use with Byzantine nodes, Celeste takes extra measures to prevent malicious nodes from jeopardizing the system. One easy attack on the deletion algorithm can be realized by a malicious node M as follows: M retrieves an object O that it does not want deleted (e.g., the AObject of a file that M wishes to "protect"), and changes its deletion token hash. It then publishes the modified object, and causes correct nodes to store it. Subsequently, if O is scheduled to be deleted, its modified version will not be deleted from correct nodes that hold it, since the correct deletion token cannot be validated against the bogus deletion-token hash. Thus, correct nodes will continue to store copies of O permanently. In order to prevent such attacks, our implementation uses verifiable deletion tokens. It is important to note that this mechanism only overcomes attacks by nodes distinct from the object's root. If a malicious node succeeds in becoming an object's root, then it can still effectively prevent the deletion process from propagating to correct nodes that hold the object.

One way to realize verifiable deletion tokens in self-verifiable objects, where the object's GUID is the hash of its data contents, is to make sure the deletion-token hash is also part of the GUID. That is, $GUID = H(data|H(deletiontoken))$, where H is a hash function, e.g., SHA-256, and | is the concatenation sign. Now, forging a deletion-token hash changes the GUID of the object.

However, since correct nodes only store objects they can validate, how can deletion objects be validated? In essence, a simple deletion object D_O only carries the deletion token used for deletion, and has the same GUID as the object O being deleted. If we have O, we can hash D_O's deletion token and compare it to O's deletion-token hash to verify D_O's validity. The problem is that the root $Root_O$ of O and D_O receives the publish message for the first copy of D_O, and must verify D_O before sending deletion requests to all the nodes holding copies of O. If $Root_O$ did not previously receive any publish message for O, it cannot verify D_O. In that case, it can either save D_O in a limited "unverified" list, or quietly discard it – it will be published again anyhow. If $Root_O$ does have some backpointer to O, then it knows O's deletion-token hash, as it is part of the meta-data sent in a publish message. $Root_O$ can then verify D_O and continue.

But this solution assumes $Root_O$ can trust the publish message for O. When $Root_O$ receives a publish message for O, it gets O's GUID, and the deletion-token hash, but it does not receive O's data contents. Consequently, $Root_O$ cannot verify the binding $GUID = H(data|H(deletiontoken))$ and must blindly

trust the publisher. To allow the root to verify publish messages and deletion objects in all situations, self-verifiable GUIDs can be calculated as $GUID = H(H(data)|H(deletiontoken))$, and the meta-data portion of all objects, including deletion objects, should also include $H(data)$, i.e., the hash of the object's data contents. For a deletion object D_O, this is the hash of the deleted object O's data contents.

Finally, AObjects are mutable, and hence cannot be self-verifiable; they are verified using digital signatures. Naturally, the AObject's deletion-token hash should also be signed, so it cannot be fabricated.

Optimization for large files or small networks. When deleting a file, messages containing the deletion token are sent to all nodes involved in the deletion process. If the number of objects related to the file to be deleted exceeds some threshold in the order of $\frac{n}{\log n}$, where n is the total number of nodes in the system, a simple broadcast of the deletion token may be better in terms of network load, and perhaps even latency. The threshold is based on the expected propagation time (in hops) in the overlay network.

Garbage collection. To free space consumed by deletion objects, it is possible to set an expiration time for every object in the system, including those that are not deleted. All objects belonging to the same file have the same expiration time. Nodes can safely garbage collect (remove) an object from their local store if that object's expiration time has passed. For every object associated with a file, the object's expiration time can only be set and modified by the file's creator, and only if the file has not been deleted yet. The refresher maintains sufficient object copies in the system only if the relevant file's expiration time has not passed yet. Thus, deleting a file with an expiration time set ensures that eventually all objects of that file will be garbage collected. However, since all state is soft, removing all deletion objects for a file will allow malicious nodes to reintroduce the deleted file to the system after the expiration time passes.

7 Conclusions

We have presented an algorithm for complete deletion of files (tier two deletion – including data obliteration) in a peer-to-peer storage system. The complete deletion of a file in such a system is an intricate operation, as it involves deleting multiple replicas of many (data and meta-data) objects, which may not all be available at the same time.

We have rigorously proven our algorithm's correctness in eventually stable runs. Moreover, we have demonstrated, using simulations, that the algorithm always succeeds to delete multi-object files entirely,

even in highly dynamic fault-prone environments. Finally, we have discussed some practical issues that arose when implementing the algorithm in Celeste, a working peer-to-peer storage system developed at Sun Microsystems labs.

A Proofs of Algorithm Correctness

In this section, we provide proofs for the lemmas and theorem of Section 4.

Lemma 1. *Let P be a partition that is eventually stable from time T, and let O be some object in the system. If after time T a live node in P has the deletion object for O, D_O, then all nodes holding (undeleted) copies of O in P eventually receive a deletion request for O.*

Proof. Let us examine partition P after time T. P is stable, so there is a node, $Root_O$, that is the root of object O, and will continue to be O's root forever. Since the deletion object D_O is in P and P is stable, $Root_O$ receives periodic publish messages for D_O. Let N be some node in P that holds a copy of O. Since P is stable, $Root_O$ also receives periodic publish messages for O from N. Consider the moment where $Root_O$ receives a publish message for O from N. If O is already marked as deleted, $Root_O$ immediately sends a deletion request for object O to N, and we are done. If not, then $Root_O$ saves a backpointer to N. When $Root_O$ next receives a publish message for D_O, it sends deletion requests to all known backpointers, since O is not marked as deleted. Since the backpointer pointing to N is still valid (due to the reliability of communication in each partition), N receives a deletion request for O. □

Lemma 2. *Let V be some version of file F, and let $VObj$ be the VObject representing this version. If the following holds for partition P: (1) P is eventually stable from time T; (2) P fulfills the complete-deletion prerequisite for file F at time T; and (3) A live node in P holds a copy of the deleted $VObj$, D_{VObj}, at time T, then all objects in P associated with version V and all earlier versions of F are eventually deleted.*

Proof. Let us examine P at time T, i.e., when P is stable and fulfills the complete-deletion prerequisite for file F. From Corollary 1 we get that all copies of $VObj$ are eventually deleted. We need to show that the corresponding BObjects and previous versions are also deleted. From Lemma 1, we get that all nodes in P holding (undeleted) copies of $VObj$ receive a deletion request. However, there might be no such nodes.

Consider first the case where there is at least one node, N, holding an undeleted copy of $VObj$ in P. When N receives a deletion request for $VObj$, it holds a local copy with a null deletion token, and hence calls *deleteVObject*, where N creates deletion objects for all the BGUIDs that are part of version V. From Corollary 1 we get that all BObjects are eventually deleted from P. Next, N calls *deleteVObject* for the previous version of the file, $V - 1$, if one exists, where it stores a copy of the corresponding deleted VObject. Applying the lemma inductively for $V - 1$ completes the proof.

Now consider the case where there is no node holding an undeleted copy of $VObj$. By the complete-deletion prerequisite, there is some node N in P to which $VObj$ is assigned. As N no longer holds an undeleted copy, it must have deleted its copy of $VObj$ by calling *deleteVObject* for it in the past. When N did so, it created deletion objects for all the BGUIDs that are part of version V as well as for the VGUID of version $V - 1$, if it exists, and stored them locally. Since N still holds these objects, and since $N \in P$, we get that, as in the previous case, all BObjects are eventually deleted (by Corollary 1), as are all the previous versions (by induction). □

Theorem 1. *Let F be a file, and let $AObj$ be the AObject representing F. If the following holds for partition P: (1) P is eventually stable from time T; (2) P fulfills the complete-deletion prerequisite for file F at time T; and (3) A live node in P holds a copy of the deleted $AObj$, D_{AObj}, at time T, then all objects in P associated with file F are eventually deleted.*

Proof. Let us examine P at time T, when P is stable. By the complete-deletion prerequisite, there is a node N in P that stores or has a some point stored the up-to-date version of $AObj$.

Consider first the case where N still holds an undeleted copy of $AObj$. Since D_{AObj} is in P, by Lemma 1 we get that N receives a deletion request for $AObj$. Since N holds an undeleted local copy of AObj, it calls *deleteAObject* for it, where it calls *deleteVObject* for $AObj$.lastVGUID. This, in turn, creates a deletion object for F's last version (since N's copy of $AObj$ is up-to-date), and stores it locally at N, which is a member of P. This satisfies the conditions of Lemma 2, and we get that all objects associated with all versions of file F are eventually deleted from P. Additionally, by Corollary 1, all copies of $AObj$ are also eventually deleted from P. Thus, all objects in P associated with file F are eventually deleted.

Now consider the case where N no longer holds an undeleted copy. In this case, N must have deleted

its copy of $AObj$ by calling *deleteAObject* for it in the past. When N did so, it created a deletion object for $AObj$.lastVGUID, and stored it locally. Since N still holds this object, and $N \in P$, we get that, as in the previous case, all objects associated with all versions of F are deleted. $\square$

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[2] G. Caronni, R. Rom, and G. Scott. Maintaining object ordering in a shared p2p storage environment. In *3rd International IEEE Security in Storage Workshop*, December 2005.

[3] I. Clarke, S. Miller, T. Hong, O. Sandberg, and B. Wiley. Protecting freedom of information online with Freenet. *IEEE Internet Computing*, Jan-Feb 2002.

[4] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS 2000*, November 2000.

[5] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[6] Z. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. Rubin. Secure deletion for a versioning file system. In *Proceedings of the Fourth Usenix Conference on File and Storage Technologies FAST 2005*, San Francisco, December 2005.

[7] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32, 1999.

[8] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, 2002.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[10] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of FAST 2003*, March 2003.

[11] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[12] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*. To Appear.

[13] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), January 2004.