J. Parallel Distrib. Comput. 69 (2009) 613–622

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Deleting files in the Celeste peer-to-peer storage system

Gal Badishi [a,*], Germano Caronni [b], Idit Keidar [a], Raphael Rom [a,b], Glenn Scott [b]

[a] Department of Electrical Engineering, The Technion – Israel Institute of Technology, Israel

[b] Sun Microsystems Laboratories, United States

ARTICLE INFO

*Article history:*
Received 8 February 2007
Received in revised form
16 October 2008
Accepted 13 March 2009
Available online 28 March 2009

*Keywords:*
Peer-to-peer
Storage
Fault-tolerance

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 contents, data and meta-data, in the aforementioned complex scenarios. The algorithm is fully functional and has been successfully integrated into Celeste.



## 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 Microsystems', 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

* Corresponding address: Department of Electrical Engineering, Technion – Israel Institute of Technology, Haifa 32000, Israel.
*E-mail address:* badishi@ee.technion.ac.il (G. Badishi).

0743-7315/$ – see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2009.03.003

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 Fig. 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 retrieval. 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