Maintaining Object Ordering in a Shared P2P Storage Environment

**Germano Caronni, Raphael Rom,
and Glenn Scott**

Maintaining Object Ordering in a Shared P2P Storage Environment

Germano Caronni
Raphael Rom, Glenn Scott

SMLI TR-2004-137 September 2004

Abstract:

Modern peer-to-peer (P2P) storage systems have evolved to provide solutions to a variety of burning storage problems. While the first generation provided rather informal file sharing, more recent approaches provide more extensive security, sharing, and archive capabilities.

To be considered a viable storage solution the system must exhibit high availability and data persistence characteristics. In an attempt to provide these, most systems assume a continuously connected and available underlying communication infrastructure. But this is not necessarily the case because equipment failures, denial of service attacks, and just poor (yet common) corporate network design may cause discontinuities and interruptions in the communication service. Any proposed storage solution needs to address such issues transparently.

Storage archival systems can live with discontinuities, as long as the stored data can be uniquely identified. Continuous update systems that allow updating data by multiple writers have harder problems to overcome since the ordering of updates needs to be maintained independently of connectivity conditions. In this paper, we propose a solution for maintaining the ordering even under severe connectivity disruptions, allowing the system to continue functioning while connectivity is disrupted, and to recover from the disruption smoothly when connectivity is restored.

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email addresses:
germano.caronni@sun.com
raphael.rom@sun.com
glenn.scott@sun.com

© 2004 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

Maintaining Object Ordering in a Shared P2P Storage Environment

Germano Caronni, Raphael Rom, Glenn Scott
Sun Microsystems Laboratories
April 2004

1 Introduction

In recent years a number of distributed storage systems have been developed, mostly in the context of peer-to-peer systems. The first generation of these, exemplified by Napster, Gnutella, and others, are essentially 'read only' systems placing little emphasis on availability and reliability but rather on general connectivity and name (directory, search) management. The second generation of such systems, exemplified by Farsite[1], Freenet [2], Oceanstore[3], Past[4], and others have been trying to solve some of the outstanding issues of the first generation systems.

One way to classify distributed storage systems is by distinguishing between archival-only and continuous-update systems. Archival-only systems (e.g., Venti[5], Freenet[2]) assume that each item stored by the system is unique and completely independent of any other item stored in the system. These systems provide mechanisms to reliably and securely store and retrieve data. Continuous-update systems (e.g., Oceanstore[3], FarSite [1], Ivy [6]) are a step closer to traditional file systems and provide, in addition, the ability to handle shared write operations, and maintain some relation between stored items. In particular, a basic assumption of these systems is that items may be updateable, and that the system must therefore maintain the notion of ”the latest version,” and possibly maintain the history of the evolution of an item so that some (or all) earlier versions can be accessed as well. Since item updates may originate simultaneously from multiple sources, the system must include a *Serializer* function that is responsible for enforcing strict ordering of these updates (such a function can be implemented in many ways from centralized to distributed).

In keeping with common practice in describing the characteristics and operation of contemporary distributed storage systems, we will refer to the items or data stored as "objects" in the remainder of our description.

All of these storage systems rely on a Distributed Object Location and Retrieval mechanism, and are generally referred to as DOLR systems [7]. DOLR systems implement an overlay network on top of the basic IP network, and each constructs its own naming and routing scheme often with Distributed Hash Tables (DHT). Typical DOLR systems are Chord[8], Tapestry[9], and Pastry[10]. The correct functioning of these mechanisms is heavily based on the availability and full-connectivity of the underlying communication infrastructure. But this cannot be always taken for granted. Failure of a critical device (e.g., router or firewall), denial of service attacks, and even storage that is occasionally disconnected from the network are but few examples.

Archival-only systems would probably work properly under disrupted or intermittent connectivity circumstances. As long as unique object names can be created, new objects can be created and stored and all objects can be retrieved subject to the connectivity constraints. When complete connectivity is restored, the system's integrity is restored as well. The independence of the stored objects is an asset here.

Continuous-update systems that also support shared writes require additional mechanism to maintain their integrity under intermittent connectivity. In particular, "the latest version" may not be consistent throughout, and separate object evolutions require special attention.

In this paper, we tackle some aspects of the problem in detail. The mechanisms we describe are general but in order to avoid inventing a completely new terminology we adopt the terminology of Oceanstore[3]. The rest of the paper describes the setting and problem in detail and proposes a scheme that allows its continuous and correct operation under adverse circumstances.

2 The Setting

Our example storage system is called *Celeste* and is generally based upon the concepts described in Oceanstore[3], with a Plaxton[11] based DOLR system. As in Oceanstore, each identifiable element is named by a Globally Unique Identifier (GUID). This covers elements such as stored data, a storage device or computer participating in the DOLR, or meta-data about stored data. While Oceanstore describes the responsibility of ordering

object updates as one of several responsibilities of an Inner Ring, we generalize and separate that specific responsibility into what we call a *Serializer*.

2.1 General

Celeste is a distributed object storage system composed of users (clients), nodes (storage devices and computers), and other resources (collectively referred to as *elements*) which are generally connected by an internetwork. Some nodes are used for object storage while others are used for message passing and management only, but none of the nodes necessarily trust one another. All element names (GUIDs) are assumed unique, are taken from a single name space, and are independent from the element location. An overlay network is constructed on top of the basic internetwork infrastructure which enables the connected nodes to reach one another, locate objects via their respective GUIDs and retrieve them. Knowledge of the GUID is sufficient to reach the named element.

We adopt a Plaxton[11] based DOLR scheme whose basis is that for any object, at any one time, there is a live node (called the *root* of the object) ultimately responsible for the object in the sense that it must know where the object is actually located. Other nodes may cache that information, for performance reasons, but the ultimate responsibility lies with the root. In a realistic environment a node serving as the root of an object may fail, and, due to the way the scheme works, another node becomes the root and is responsible for that object. Various implementations go to great lengths to ensure that the duration of this fail-over transient state is as short as possible.

Nodes participating in the DOLR network maintain state information to facilitate or enable their proper operation (some of which they share with one another). To ensure that a DOLR network does not have a single point of failure, all state information should be soft, namely recoverable by the node should it lose its state information for whatever reason. To maintain the soft state information, the source of that information must retransmit it from time to time. Obviously, because of the resources that are required, there is a tradeoff between the frequency of this retransmission and the time it takes for a node to recover lost state information. Some of the mechanisms described in this document are aimed at reducing this recovery time. Almost by definition, they are not completely failsafe in that there are circumstances in which they do not work. While these should be rare, the standard retransmissions are always the fallback procedure. In general, we do assume that the DOLR mechanism functions properly subject to general connectivity constraints.

Celeste considers the objects as evolving, i.e., every object consists of a sequence of updates. To maintain the order, objects are assigned, in addition to their name, a version

number, and every object is linked to the object it updates. The object at the head of this linked list is the latest version. (If version numbers are assigned sequentially, then the object with the highest version number is the latest version). For this to hold true, the system must maintain strict ordering of the updates which is done by a *Serializer* function. The Serializer can be implemented in a distributed fashion by means of a Byzantine Agreement among a set of nodes[12] (called an Inner Ring in Oceanstore[3]). To be consistent, the same Serializer must serve all updates (versions) of an object although, obviously, one Serializer can serve multiple objects.

2.2 Object Ordering Problem With Intermittent Connectivity

Virtually all the systems mentioned earlier assume that the infrastructure is richly connected, so that when a route fails, another one can be devised by bypassing the troubled area. This, however, cannot be relied upon as, for example, a corporate intranetwork is typically not that richly connected, DoS attacks may render parts of the network inaccessible, and in some cases a deliberate disconnect is imposed on the system (say, for protected computation). The question is whether a storage system using a DOLR network thus described can, under such circumstances, maintain the specified service.

We consider the case where after working properly for some time the DOLR infrastructure becomes *partitioned* and no element in one partition can access or be accessed by an element in another partition. The partitioned DOLR network actually becomes several separate ones and the Celeste is now a collection of individual Celestes with portions of objects arbitrarily spread around the partitions. Because object evolution, as presented above, is not well defined, it is necessary to refine the definition.

The most natural solution to the problem of handling partitioning is that each partition continues the evolution of an object independently from the other. When the partitions reunite, the Celeste will merge the different evolutions and generate a single one again but, while partitioned, the “latest version” may depend on the partition being accessed. Note that independent evolutions are unavoidable since it is impossible to distinguish between a temporary partition and a permanent loss of nodes. In this scenario, the toughest case is probably that of two processes of an application running on two different machines communicating with each other but each getting a different view of the Celeste (see Figure 1).

Our definition of object evolution is that the latest version is the highest version available in the partition in which it is requested, and different processes, through their private communication, can determine whether or not they are viewing different evolutions of the object. Beyond guaranteeing object evolution in this vein, the Celeste guarantees that

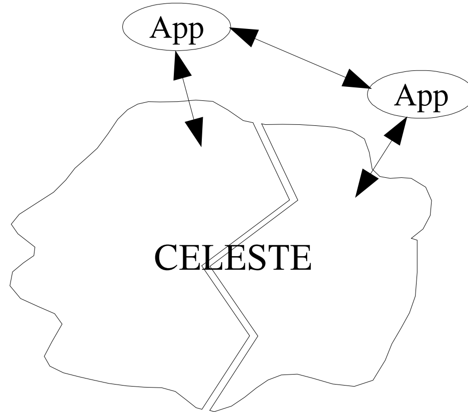

Figure 1: Distributed application accessing a partitioned Celeste

when the partitions reunite, eventually a single view of the object is restored, without loss of any evolutionary branches.

The approach we propose here is quite different from that implemented in the Bayou[13] system (other than the general structure), in that the Celeste does not undertake to resolve object mergers, or behavior in a partitioned environment, but rather to maintain a consistent object evolution and present to the application all the information it needs to perform its own consistency management.

3 Object Evolution Under Partition

In general, different policies to control object evolution in the face of partitioning should be supported by the Celeste. Most of these policies are easily implemented, such as freezing the object evolution until the Serializer comes back, or allowing only currently operating Serializers to continue operating. It is when one tries to allow for independent evolution in different partitions, and reconcile this state later, that the hard problems arise. In the present section our solution to this problem is discussed in depth.

To control object evolution in the face of a network partition, we introduce the notion of a generation (sometime referred to as incarnation) which is a period during which the underlying DOLR network does not encounter any major event. A *major event* is defined as partitioning of the network or the unification of (previously partitioned) network partition (in view of a certain Serializer). Every generation is identified by a unique generation number. In Celeste, generation numbers are selected such that their uniqueness is guaranteed with high probability. New generation numbers are created randomly based on the

lowest GUID of the nodes that comprise the Serializer along with a sequence number maintained at that node. When a Celeste is partitioned, each of the partitions will choose a different generation number to allow the independent evolution of objects it manages.

As described earlier, every object has a name and version number. To accommodate the separate evolution of objects in disjoint partitions we add the generation number to the object description. In terms of version sequencing, the first version of a generation is linked to some version of a previous generation which it follows. Subsequent versions will then be linked normally to the version they update within the same generation (on occasion, and for backtracking and audit purposes, an object will be additionally linked to an object of a previous version in a different generation. See the example below for explanation).

There is a slight argument to be made for starting version numbers as '1' whenever a new generation starts. The purpose would be to allow enumeration of specific versions (such as "the first version"), without having to follow the chain. However, any sequencing of version numbers will do.

Translated into this terminology, the semantics of Celeste object ordering is that the latest version is the highest version number of the most current generation number of the object of which the node is aware.

3.1 The Role Of The Serializer

The Serializer mentioned earlier plays a critical role in object updates. In fact, the name (as a GUID) of the Serializer is stored with every object, so that when an object is updated the correct Serializer, i.e., that which is responsible for the object being updated, is contacted. To be specific, every Serializer is described by an object (S_{id}) that contains minimal but necessary information about the Serializer; among them are the GUIDs of the nodes that implement it and the current generation number of the Serializer. The GUID of that object is referred to as the Serializer's *name*, and is assigned when the Serializer is created and remains unchanged throughout the life of the object. The pair <*Serializer-name, generation number*> is unique, i.e., refers to a single Serializer. However, it is possible that the same Serializer is associated with many such pairs. The S_{id} object itself is stored in each of the Serializer nodes, at the root of the object (see section 4 below), and is widely publicized. When the membership of the Serializer changes by consent (i.e., enough of its members are still accessible) the name of the Serializer and the generation number remain the same but the content of the S_{id} changes to reflect the changes in the membership of the nodes that implement it.

When a user's object needs to be updated, the updating node sends the update transaction, including the version and generation of its origin, to the S_{id}. The update will end up at the Serializer since this is where the S_{id} is stored. Note that the content of the S_{id} is needed for the construction of the Serializer and not for its operation. When the content of the S_{id} needs to be changed, the Serializer will generate a new such object (with the same name and GUID), and disseminate it to all interested parties.

The S_{id} object is an example of an object handled by the Celeste for its own management and operations activities. Unlike user objects, which are generated by the user, the S_{id} is generated by the Celeste itself, and is part of the (soft) state of the Celeste. We dwell on this in more detail in section 4.

3.2 The Evolution Of The Serializer

A Serializer can decide to transfer its duties to another Serializer (voluntary Serializer reassignment). A special case of a voluntary reassignment is the de-commissioning of a Serializer, which occurs when the Serializer wants to reassign all the objects it is responsible for to another Serializer. This can be done easily by having the new Serializer assume the name and generation number of the old Serializer (in addition to its old name), and continue smoothly thereafter. This requires the old Serializer to unpublish the S_{id} it stored and the new Serializer must store and publish the S_{id} of the old Serializer with its contents changed to reflect the new membership. A more complicated case to handle is that of an involuntary Serializer reassignment which happens when the Serializer is incapacitated (e.g., more than a half of its members are inaccessible). When this happens, a new Serializer must be created, or the duties reassigned to another Serializer. This new Serializer will assume the name of the old Serializer (which is recorded in any object it controls) but with a new generation number.

Involuntary Serializer reassignment may stem from awkward situations. The problem scenario of Celeste partitioning is severe because communications between the various partitions is not available. When a Celeste is partitioned in two (the most typical case), one partition is guaranteed to have a working Serializer which will retain its generation number until the two partitions are reunited (this case is depicted in Figure 2). A more severe scenario arises, when, for a given object, a Celeste is partitioned into more than two pieces. In this case at most one partition (quite possibly none) will have a working Serializer. What this means is that a new Serializer will be generated in every partition, all having the same name and each having a different generation number. Objects in each partition will evolve independently until the partitions are reconnected. At reconnect time, the Serializers must be merged (or one must take over for the others) and the generation number changed.

So why do we need a pair of identifiers (name and generation number)? Generation numbers track independent object evolutions due to partitioning and Serializer incapacity and provide a unique name to every object version. Serializer names are needed to identify the Serializer that is assigned to an object, to identify the occurrence of independent evolution, to simplify the merge. They also facilitate the reassignment of Serializers so it can be done not on an object-by-object basis but rather on a Serializer-by-Serializer basis. Generation numbers are dynamic; Serializer names are static.

3.3 An Example Of Object Evolution

Figure 2 depicts an evolution of an object over time. An object is described by the tuple <*object name, version, generation, Serializer name*> where an object name, version, and generation number are as previously explained, and Serializer name is the name of the Serializer handling the object. The Serializer itself is described by the tuple <*Serializer name, generation number, Current, previous1, previous2*> where *Current* is an indicator whether this Serializer is the latest known version, and where *previous* points to the previous generations that this Serializer follows. For ease of deciphering, current Serializers are in blue and previous ones in red (color coding is just for convenience). Time is indicated in some artificial units and is for reference only. Objects with the same color belong to the same generation.

At time 10 an object whose name is 80 is created; this is version 1 and it is assigned Serializer 10 whose generation number is 1 and is the current Serializer (the blue circle). At times 20, 30, and 40, updates occur which create objects that differ from the previous one only by the version number. At time 50 the Celeste is partitioned with Serializer 10 remaining intact in the right partition. The left partition realizes there is no Serializer 10, and thus creates a new generation, number 2 (with green triangles), while recording the data of the old one. (Note that all this could occur due to an event not necessarily related to object 80, for example related to another object also controlled by Serializer10; in section 6 we describe how this actually happens). Subsequently, we notice two separate evolutions of object 80.

At time 140 the two partitions unite. The reconstitution is manifested by the unification of the two S_{id}s. When this occurs, a new Serializer 10 is created (from one of the old ones) which gets a generation number 4 (with red squares), becomes the current one, and points to the previous generations 1 and 2. This unification is independent of object 80 and effects all objects handled by Serializer 10. The unification process is triggered by a node watching the S_{id} object and noticing a non-matching generation number (see detailed explanation in section 6 below).

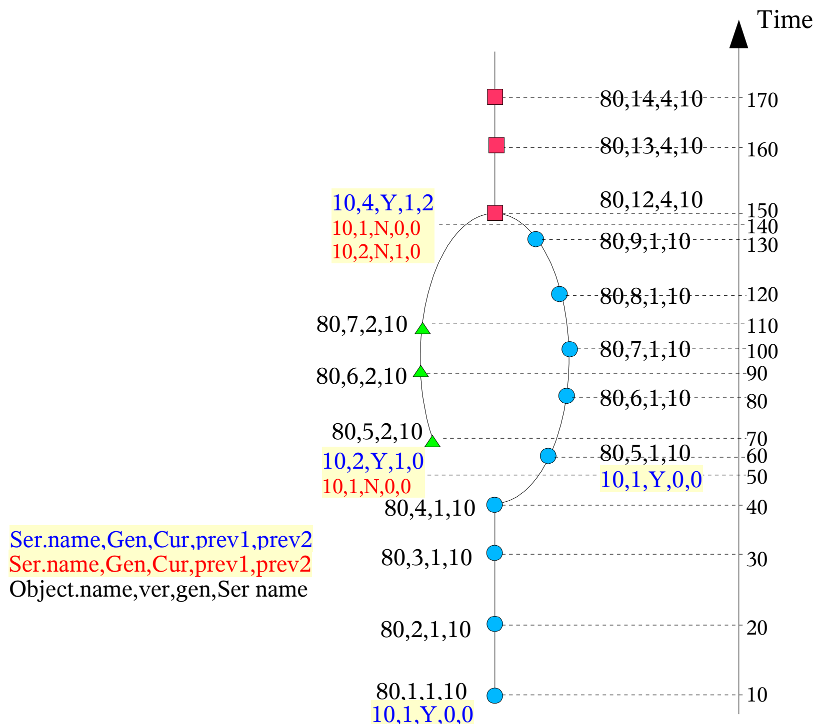

Figure 2: Timeline of an object evolution

At time 150 a new update to object 80 takes place, for example, by a user updating object <80,7,2,10>. Serializer 10 realizes that an object from a previous generation is being updated and determines that the new object is the current one, and assigns it a version number (actually arbitrary, 12 in the example). It can now point to the last known object of generation 2 as the previous version. At time 160 another update occurs, this one updating object <80,9,1,10>. At this point (the existent) Serializer 10 realizes that an old generation is being updated , assigns it version 13, and points to both the previous numerical version (object <80,12,4,10>) and the one it updates (object <80,9,1,10>).

With the pointers thus recorded, it is possible to trace the entire evolution of an object from any point backwards. Note also that during the entire process neither the object name nor the name of its Serializer are changed, greatly simplifying the management.

4 Maintaining Serializer State

For the Celeste to work properly, a client wishing to update an object must be able to reach the Serializer of that object. To do so, the client will send a message to the S_{id} ob-

ject (whose name appears in the object being updated) along with the updated object. Because a copy of the S_{id} object is stored in each of the members of the Serializer, this message will arrive at one of those members of the Serializer and be properly handled. Under normal circumstances this is a straightforward operation, but not so when the Celeste becomes partitioned. In the rest of this section we define how the various necessary relations are maintained. While we do this with the Serializer example, it is a general mechanism useful for maintaining soft-state.

Consider a Celeste client C attempting to update a general Celeste object O, whose Serializer S is described by the object S_{id}. Let S_{or} be the 'old' DOLR root of S_{id}, i.e., S_{or} knows the whereabouts of S_{id}, namely S, before the Celeste partitions. When the Celeste is partitioned, client C will be in one of the following states with respect to object O's Serializer:

- Intact -- The client node can reach both S, object O's Serializer, and S_{or}.
- Broken -- The client node cannot reach S, but can reach the S_{or}.
- Orphaned -- The client node can reach S, but cannot reach S_{or}.
- Vacant -- The client can reach neither S nor S_{or}.

Figure 3 depicts these relations. The way the DOLR network works, it will always find a root for object S_{id}; so let S_{nr} be the root node of object S_{id} after the Celeste partitions. In the intact and broken state this will be S_{or}, and in the orphaned and vacant states it will be some other node (not shown in the figure).

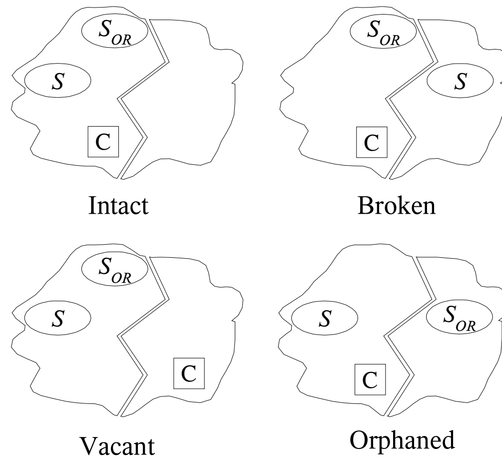

Figure 3: Client state w.r.t. Serializer after Celeste partition

Clearly when in the intact state that portion of the Celeste can continue functioning as if a partition did not occur (the right branch in Figure 2 is such an example). In any of the

other three states some action must be taken to bring the Serializer to a working condition. We assign S_{nr} the responsibility to coordinate the reconstruction.

The S_{nr} becomes aware of the partitioning when messages destined to the Serializer start arriving at it (because the S_{id} is widely publicized, messages destined to it rarely reach its root). The S_{nr} then searches for the Serializer, and if found (in the orphaned case) proceeds normally without changing the generation number (again, the right branch in Figure 2 can be such an example). Otherwise it must trigger the Serializer construction process, assign the new Serializer the old name with a new generation number and then proceed as usual. Note that, for efficiency reasons, as many nodes as possible that participated in the old Serializer should be part of the newly constructed one; the GUIDS of these nodes are listed in the S_{id} object. Note also that a single failure may cause the Celeste to partition to multiple partitions, most of which will be vacant.

The only issue in the above description is locating the Serializer (i.e., the S_{id} object) by S_{nr}. The S_{nr} knows the name of the Serializer and its GUID but the attempt to send it a message will fail as the message will promptly return to the S_{nr}, since it is ultimately responsible to know where the S_{id} is. Clearly, the S_{nr} can wait until the S_{id} is republished, but this might take a long time if the Serializer exists at all in the current partition. To overcome this problem we devise the following mechanism to maintain soft-persistent objects containing the soft state of the S_{id} object.

To enable a root of a soft-persistent object to locate and retrieve its contents, we store *shadows*, i.e., identical copies under different names in such a manner that one name can be derived from the other. Creating $name_1$ $name_2$, etc., from *name* is but one example. Then by using the normal hashing, the GUIDs of the shadows are created. We require the roots of the shadow objects to store the object themselves.

A root of a soft-persistent object that needs to retrieve the contents of the object will try to reach one of the shadow roots storing those objects. Because of the randomness of the hashing, the roots of the shadow objects will be topologically disparate from one another. With enough copies, a shadow root is likely to be in the same partition.

One could make the argument that in the unlikely case when no shadow root and no member of the Serializer survive in a partition, then that partition is most likely too small to be viable in regards to this object. Its history and actual data are most likely not reachable as well.

The initial keeper of the soft-persistent object publishes the object normally and when the root of the object discovers its role, it makes a local copy and publishes the object under

the shadow names $name_1$ $name_2$, etc. The root of the *i*-th shadow, upon realizing its role, will make a local copy and and publish the object under its original name. By tracking the (re)publication of the various shadows, the original root can verify that enough copies are indeed maintained. Note that this star implementation places more responsibility on the original root than on the shadows; a more balanced implementation would be to use a ring whereby the *i*-th shadow publishes the object under the *i+1*st name.

5 Maintaining Serializer Identities

In the previous sections, we described a mechanism that necessitates the creation (and disbanding) of Serializers depending on connectivity changes in the underlying DOLR network. Several security issues surface in this context. In this Section we sketch out some of those issues, and discuss remedies. In particular we will concentrate on issues of continuity, once an object has been linked to a Serializer. This linking does not imply how the trustworthiness of a Serializer is determined, and how it was actually chosen; the problem of trust management is outside the scope of this document.

First of all, one has to keep in mind that a Serializer not only decides in what order changes to objects take place, but also if they take place at all. Thus, the Serializer as a whole needs to be trustworthy to the owner of a specific object, as well as to the writer and later readers of the object. Second, with the Serializer enforcing update ordering, so far nothing prevents it from issuing updates to the object on its own. Third, Serializers may become inoperable due to node failure or network partitioning. Finally, Serializers can be attacked by others to make them unavailable to legitimate users. One has to consider that the name of the Serializer can never change. Thus a non-functional but existent Serializer can disrupt normal operations on the objects for which it is responsible.

Linking an object to a Serializer is done by the owner of an object, at object creation time. After selecting an existing Serializer (or causing the creation of a new one), the owner takes the name of the Serializer (its GUID, same as the name of the S_{id}) and signs it with the key that corresponds to the GUID of the owner. Since the name of the Serializer is in fact derived from its public key, this is a verifiable binding. All operations of the Serializer on the object will carry a signature, done with the related private key. That key is divided up, and distributed among all members of the Serializer, such that at most two thirds of the collaborating members are needed to issue a signature.

The Serializer signs every update it processes. That signature is widely verifiable, and in particular allows to prove the authenticity of the update from the Serializer's perspective. Additionally, all updates are also signed by the actual writer. Serializers may (but don't necessarily have to) validate this signature by the writer. Given that updates are bound to

writers, Serializers cannot issue update requests of their own, if readers verify the integrity and authenticity of the latest version.

The main problem of having Serializers survive network partitioning is twofold. The availability of the Serializer public data needs to be assured (this is done by making the S_{id} highly available, and self-maintained, as explained in the previous Sections), and the private data of the Serializer (e.g., its private key, that binds to the name of the S_{id}) needs to be available to whatever legitimate incarnation(s) of the Serializer in multiple partitions.

Clearly, as long as at least half of the members of the Serializer are reachable, and as long as the number of malicious members is small enough, the Serializer as a whole can function. Specifically, it can elect new members, and provide them with their share of the private key. However if an insufficient number of members are available, or if even no former member of the Serializer is reachable, then a new valid Serializer must still be able to emerge out of the void.

Without considering security, this is achieved by making the S_{id} highly available. If at least one copy of the S_{id} can be found, then the Serializer can be bootstrapped if needed. Our solution described in Section 4 takes care of assuring availability by electing random nodes as shadow roots, that then keep the S_{id} replicated and alive. Unfortunately, one can not simply hand out the private key of the Serializer in the S_{id} (or to the shadow roots, for that matter) because otherwise anyone having access to it could impersonate the Serializer quite easily. Consequently, the private key itself needs to be stored in a distributed and secure manner. The most natural approach is to divide the private key into separate shares, and then have some entities store those shares. The following classes of entities come to mind, and can all be used for the purpose of increasing the chances of recunstructing the private key:

1. Within the Serializer members
2. With the shadow roots
3. With a set of random or selected highly-available nodes

The private key of the Serializer can even be generated in a distributed manner, e.g., by following the algorithm outlined by Boneh et al.[14], possibly enhanced along the lines of [15]. One would deploy a sharing scheme where different proportions of these three classes of share holders are necessary to reconstruct the shared private key of the Serializer. One example would be to require at least 6 of the, say, 10 Serializer members, where each Serializer member is replaceable by three out of twenty shadows, each of which is replaceable by, say, 5 out of 1000 random nodes. In effect, a reconstituted Serializer could then consist of one shadow root, plus 85 of the random nodes. Obviously, it would

quickly allocate a much smaller number of regular members, which would then perform all future computation.

One further consideration is needed for the situation where the partition is formed such that reconstitution of the Serializer key is not possible. In that case, serialized updates to its objects become impossible, or alternatively, a Serializer with a different key is created, and a new branch of the object evolves from this. Reunification of branches can then be done at a later point in time, this being an application-dependent operation. However, one should note that in the event of the formation of such a small partition, it is highly likely that the partition is dysfunctional in any case, since simply not enough data fragments, AGUIDS, etc., are available.

6 Putting It All Together

In the previous sections we showed how objects can evolve separately when the Celeste is partitioned and how the Serializer functions in face of infrastructure impairment. What is left to show is that these mechanisms are all put into action at the right time and right order.

The Celeste mode of operation is event driven, mostly responding to actions rather then relying on timeouts. A basic event is a node receiving a request to update an object. Because this is an update, the name of the Serializer is included in the descriptor of the object being updated (together with the generation and version number of the object from which the changes originates) and is thus available to our node. The updated object, along with the descriptor of the object being updated, is then transmitted to the Serializer. During normal operation of the Celeste, the information will arrive at the Serializer and be handled properly. However, if the Celeste has undergone partitioning, the update request will arrive at the S_{nr}, and a Serializer will be constructed as explained in section 4. When the update is complete our node will receive a final acknowledgment which includes the generation number of the Serializer. By comparing this number with the generation number originally held by our node, the client can be notified of the changes that occurred to the Celeste. This capability is notable because it permits an application to ensure that it is working on the same object over time (in fact, Celeste allows updates to be predicated on unchanged generation numbers [3]).

Consider now the case of Celeste restoration, i.e., when two partitions of a (previously partitioned) Celeste are reconnected. The problem is that for any object that evolved independently during the partition time, two Serializers with the same name are operating in the united Celeste. We handle the situation as follows. According to the rules, every Serializer publishes its own S_{id}, which includes its generation number and is treated as a soft-

persistent object. When a Celeste is united, the DOLR will assign a single node as the root of the S_{id} object (this will be one of the pre-reconnection roots). However, because there are two Serializers operating by the same name the root node will receive two different published messages of the S_{id}, with different generation numbers. This serves as a signal to the root which then disables one of the two Serializers (and which subsequently causes the disabled Serializer to unpublish the old S_{id} as well as its shadows).

From an object evolution perspective, the moment at which a Serializer is disabled (and another one takes its place) is considered the instant at which the Celeste reunited. This determination is important for the semantics of “the latest version.” Before the Serializer was disabled there had been two parallel evolutions with two different generation numbers; after the disabling, there is a single evolution thread in place. One final note is that because the object descriptor points to both the object it is updating and the one considered by the Celeste as the latest version, one could trace any object backwards to all the versions that lead to it, including all those that evolved independently, thereby enabling the application to take whatever steps it needs to construct an updated and uniform view of the object.

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 *Proceedings of the 5th Symposium on Operating Systems and Design Implementation OSDI*, (December 2002).

[2] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong, "*Freenet: A Distributed Anonymous Information Storage and Retrieval System*," Lecture Notes in Computer Science, (2009), pp. 44-46 (2001)

[3] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao, "OceanStore: An Architecture for Global-scale Persistent Storage," in *Proceedings of ACM ASPLOS*, (November 2000).

[4] Antony I. T. Rowstron and Peter Druschel, "Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility," pp. 188-201 in *Symposium on Operating Systems Principles* (October 2001)

[5] S. Quinlan and S. Dorward, "Venti: a new approach to archival storage," in *First USENIX conference on File and Storage Technologies*, (January 2002).

[6] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen, "Ivy: A Read/Write Peer-to-peer File System," in *Proceedings of the 5th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI '02), (December 2002).

[7] F. Dabek, B. Zhao, P. Druschel, and I. Stoica, "Towards a common API for structured peer-to-peer overlays," in *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, (February 2003).

[8] Ion Stoica, Robert Morris, David Karger, M. Francs Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," pp. 149-160 in *Proceedings of the SIGCOMM 2001 conference on applications, technologies, architectures, and protocols for computer communications* (August 2001)

[9] Zhao, B. Y., Huang, L., Rhea, S. C., Stribling, J., Joseph, A. D., and Kubiatowicz, J. D., "*Tapestry: A global-scale overlay for rapid service deployment*," IEEE Journal on Selected Areas in Communications, (22)1, pp. 41-53 (January 2004)

[10] Antony Rowstron and Peter Druschel, "*Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*," Lecture Notes in Computer Science, (2218), pp. 329--350 (2001)

[11] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," pp. 311-320 in *ACM Symposium on Parallel Algorithms and Architectures* (June 1997)

[12] Castro and Liskov, "Practical Byzantine Fault Tolerance," in *OSDI: Symposium on Operating Systems Design and Implementation*, (February 1999).

[13] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," pp. 172-182 in *Proceedings of the fifteenth ACM symposium on Operating systems principles* (December 1995)

[14] Dan Boneh and Matthew Franklin, "*Efficient generation of shared RSA keys*," Journal of the ACM, (48)4, pp. 702-722 (2001)

[15] Y. Frankel, P. MacKenzie, and M. Yung,, "Robust Efficient Distributed RSA Key generation," pp. 663-672 in *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)* (May 1998)

About The Authors

Germano Carroni

Germano Caronni (IEEE/ACM/ISOC) received his M.Sc. in Computer Science in 1993, and a Ph.D on QoS-based Dynamic Security in 1999, both from ETH Zurich. He was one of the first to invent a process to watermark images, participated in the IETF (IPSEC), led the independent implementation effort for SKIP (secure TCP/ IP), and its integration into an adaptive firewall. In 1997, he won the RC5/48 challenge of RSA DSI. Since late 1997, Mr. Caronni has been with Sun Microsystems, where he has introduced a novel solution to secure multicasting, worked on authentication frameworks, participated in the design of an overall security architecture for Sun's products, and co-invented the concept of Public Utility Computing. He is currently a member of the Security Research Group in Sun Microsystems Laboratories, and principal investigator on secure storage solutions.

Raphael Rom

Raphael Rom received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion—Israel Institute of Technology, Haifa, Israel, and the Ph.D. degree in computer science from the University of Utah, Salt Lake City, UT. His specialization are the areas of computer networks, distributed systems and communication systems. Within these areas his interest and expertise include architectural design, algorithms and protocols for network functionalities, and performance analysis of data communication and wireless networks. He has extensively published scientific research papers in the top journals as well as leading international conferences and served on the editorial boards of various professional journals. Raphael Rom has been with Sun Microsystems since 1989 and as a professor with the Faculty of Electrical Engineering at the Technion since 1981.

During Dr. Rom's career he was a senior researcher on the research staff of SRI International in California where he participated in the design and implementation of the first packet radio network. He held visiting positions in IBM T.J. Watson Research Center and Stanford University. During 1999-2003 he was chairman of the Israeli InterUniversity Computer Center (IUCC) in which capacity he was responsible for Israel's national academic research network, high-performance computing center, and the academic relations with the European Community. In Sun Microsystems Laboratories, he was the manager of the high speed networking group where he designed and implemented advanced ATM and wireless network architectures.

Glenn Scott

Glenn Scott has been with Sun Microsystems since 1990 working on a wide array of networking concepts, projects and products and is currently focused on secure virtual enterprise networks and secure computing in totally public environments.

Prior to Sun Labs, Scott was the Director of Engineering and the Chief Technologist for the Internet Commerce and Security business unit of SunSoft, which was responsible for defining Security and Electronic Commerce products and technologies for Sun Microsystems, Inc. Previously, Scott was a founding member of the Internet Commerce Group, an incubator-style business formed within Sun Microsystems Laboratories focusing on technology and product development for conducting business over the Internet.

Prior to joining Sun, Scott was with System Development Corporation's Santa Monica Research Center as a researcher in formal methods of software development, automatic theorem provers, secure operating systems, and trusted operating systems.

Scott is a graduate of Chapman University, with an academic background in Computer Science, Electrical Engineering, Chemical Engineering, and Applied Linguistics.