# OceanStore: An Architecture for Global-Scale Persistent Storage*

John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski,
Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea,
Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao

University of California, Berkeley

http://oceanstore.cs.berkeley.edu

## ABSTRACT

*OceanStore is a utility infrastructure designed to span the globe and provide continuous access to persistent information. Since this infrastructure is comprised of untrusted servers, data is protected through redundancy and cryptographic techniques. To improve performance, data is allowed to be cached anywhere, anytime. Additionally, monitoring of usage patterns allows adaptation to regional outages and denial of service attacks; monitoring also enhances performance through pro-active movement of data. A prototype implementation is currently under development.*

## 1 INTRODUCTION

In the past decade we have seen astounding growth in the performance of computing devices. Even more significant has been the rapid pace of miniaturization and related reduction in power consumption of these devices. Based on these trends, many envision a world of ubiquitous computing devices that add intelligence and adaptability to ordinary objects such as cars, clothing, books, and houses. Before such a revolution can occur, however, computing devices must become so reliable and resilient that they are completely transparent to the user [50].

In pursuing transparency, one question immediately comes to mind: *where does persistent information reside?* Persistent information is necessary for transparency, since it permits the behavior of devices to be independent of the devices themselves, allowing an embedded component to be rebooted or replaced without losing vital configuration information. Further, the loss or destruction of a device does not lead to lost data. Note that a uniform infrastructure for accessing and managing persistent information can also provide for transparent synchronization among devices. Maintaining the consistency of these devices in the infrastructure allows users to safely access the same information from many different devices simultaneously [38]. Today, such sharing often requires laborious, manual synchronization.

Ubiquitous computing places several requirements on a persistent infrastructure. First, some form of (possibly intermittent) *connectivity* must be provided to computing devices, no matter how small. Fortunately, increasing levels of connectivity are being provided to consumers through cable-modems, DSL, cell-phones and wireless data services. Second, information must be kept *secure* from theft and denial-of-service (DoS). Since we assume wide-scale connectivity, we need to take extra measures to make sure that information is protected from prying eyes and malicious hands. Third, information must be extremely *durable*. Therefore changes should be submitted to the infrastructure at the earliest possible moment; sorting out the proper order for consistent commitment may come later. Further, archiving of information should be automatic and reliable.

Finally, *information* must be divorced from *location*. Centralized servers are subject to crashes, DoS attacks, and unavailability due to regional network outages. Although bandwidth in the core of the Internet has been doubling at a incredible rate, *latency* has not been improving as quickly. Further, connectivity at the leaves of the network is intermittent, of high latency, and of low bandwidth. Thus, to achieve *uniform* and *highly-available* access to information, servers must be geographically distributed and should exploit caching close to (or within) clients. As a result, we envision a model in which information is free to migrate to wherever it is needed, somewhat in the style of COMA shared memory multiprocessors [21].

As a rough estimate, we imagine providing service to roughly $10^{10}$ users, each with at least 10,000 files. OceanStore must therefore support over $10^{14}$ files.

### 1.1 OceanStore: a True Data Utility

We envision a cooperative utility model in which consumers pay a monthly fee in exchange for access to persistent storage. Such a utility should be highly-available from anywhere in the network, employ automatic replication for disaster recovery, use strong security by default, and provide performance that is similar to that of existing LAN-based networked storage systems under many circumstances. Services would be provided by a confederation of companies. Each user would pay their fee to one particular "utility provider", although they could consume storage and bandwidth resources from many different providers; providers would buy and sell capacity among themselves to make up the difference. Airports or small cafés could install servers on their premises to give customers better performance; in return they would get a small dividend for their participation in the global utility.

*This research is supported by NSF career award #ANI-9985250, DARPA grant #N66001-99-2-8913, and DARPA grant #DABT63-96-C-0056.

Patrick Eaton is supported by a National Defense Science and Engineering Graduate Fellowship (NDSEG); Dennis Geels is supported by the Fannie and John Hertz Foundation; and Hakim Weatherspoon is supported by an Intel Masters Fellowship.

Appears in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2000), November, 2000

Ideally, a user would entrust all of his or her data to OceanStore; in return, the utility's economies of scale would yield much better availability, performance, and reliability than would be available otherwise. Further, the geographic distribution of servers would support *deep archival storage*, i.e. storage that would survive major disasters and regional outages. In a time when desktop workstations routinely ship with tens of gigabytes of spinning storage, the management of data is far more expensive than the storage media. OceanStore hopes to take advantage of this excess of storage space to make the management of data seamless and carefree.

## 1.2 Two Unique Goals

The OceanStore system has two design goals that differentiate it from similar systems: (1) the ability to be constructed from an *untrusted infrastructure* and (2) support of *nomadic data*.

**Untrusted Infrastructure:** OceanStore assumes that the infrastructure is fundamentally *untrusted*. Servers may crash without warning or leak information to third parties. This lack of trust is inherent in the utility model and is different from other cryptographic systems such as [35]. Only clients can be trusted with cleartext—all information that enters the infrastructure must be encrypted. However, rather than assuming that servers are passive repositories of information (such as in CFS [5]), we allow servers to be able to participate in protocols for distributed consistency management. To this end, we must assume that most of the servers are working correctly most of the time, and that there is one class of servers that we can trust to carry out protocols on our behalf (but not trust with the content of our data). This *responsible party* is financially responsible for the integrity of our data.

**Nomadic Data:** In a system as large as OceanStore, locality is of extreme importance. Thus, we have as a goal that data can be cached *anywhere*, *anytime*, as illustrated in Figure 1. We call this policy *promiscuous caching*. Data that is allowed to flow freely is called *nomadic data*. Note that nomadic data is an extreme consequence of separating information from its physical location. Although promiscuous caching complicates data coherence and location, it provides great flexibility to optimize locality and to trade off consistency for availability. To exploit this flexibility, continuous *introspective* monitoring is used to discover tacit relationships between objects. The resulting "meta-information" is used for locality management. Promiscuous caching is an important distinction between OceanStore and systems such as NFS [43] and AFS [23] in which cached data is confined to particular servers in particular regions of the network. Experimental systems such as XFS [3] allow "cooperative caching" [12], but only in systems connected by a fast LAN.

The rest of this paper is as follows: Section 2 gives a system-level overview of the OceanStore system. Section 3 shows sample applications of the OceanStore. Section 4 gives more architectural detail, and Section 5 reports on the status of the current prototype. Section 6 examines related work. Concluding remarks are given in Section 7.

# 2 SYSTEM OVERVIEW

An OceanStore prototype is currently under development. This section provides a brief overview of the planned system. Details on the individual system components are left to Section 4.

The fundamental unit in OceanStore is the *persistent object*. Each object is named by a *globally unique identifier*, or GUID.

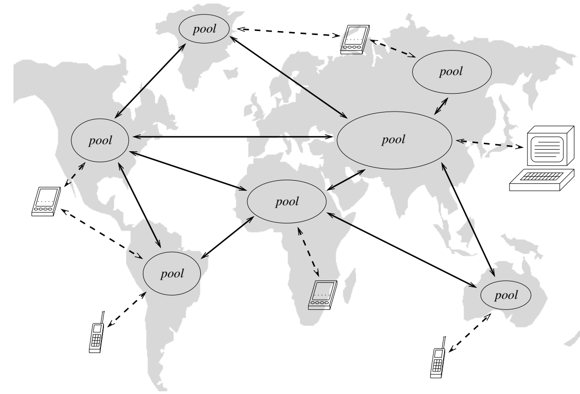


**Figure 1:** ***The OceanStore system.*** **The core of the system is composed of a multitude of highly connected "pools", among which data is allowed to "flow" freely. Clients connect to one or more pools, perhaps intermittently.**

Objects are replicated and stored on multiple servers. This replication provides availability[1] in the presence of network partitions and durability against failure and attack. A given replica is independent of the server on which it resides at any one time; thus we refer to them as *floating replicas*.

A replica for an object is located through one of two mechanisms. First, a fast, probabilistic algorithm attempts to find the object near the requesting machine. If the probabilistic algorithm fails, location is left to a slower, deterministic algorithm.

Objects in the OceanStore are modified through *updates*. Updates contain information about what changes to make to an object and the assumed state of the object under which those changes were developed, much as in the Bayou system [13]. In principle, every update to an OceanStore object creates a new version[2]. Consistency based on versioning, while more expensive to implement than update-in-place consistency, provides for cleaner recovery in the face of system failures [49]. It also obviates the need for backup and supports "permanent" pointers to information.

OceanStore objects exist in both *active* and *archival* forms. An active form of an object is the latest version of its data together with a handle for update. An archival form represents a permanent, read-only version of the object. Archival versions of objects are encoded with an erasure code and spread over hundreds or thousands of servers [18]; since data can be reconstructed from *any* sufficiently large subset of fragments, the result is that nothing short of a global disaster could ever destroy information. We call this highly redundant data encoding *deep archival storage*.

An application writer views the OceanStore as a number of sessions. Each session is a sequence of read and write requests related to one another through the session guarantees, in the style of the Bayou system [13]. Session guarantees dictate the level of consistency seen by a session's reads and writes; they can range from supporting extremely loose consistency semantics to supporting the ACID semantics favored in databases. In support of legacy code, OceanStore also provides an array of familiar interfaces such as the Unix file system interface and a simple transactional interface.

[1] If application semantics allow it, this availability is provided at the expense of consistency.

[2] In fact, groups of updates are combined to create new versions, and we plan to provide interfaces for retiring old versions, as in the Elephant File System [44].

Finally, given the flexibility afforded by the naming mechanism and to promote hands-off system maintenance, OceanStore exploits a number of dynamic optimizations to control the placement, number, and migration of objects. We classify all of these optimizations under the heading of *introspection*, an architectural paradigm that formalizes the automatic and dynamic optimization employed by "intelligent" systems.

# 3 APPLICATIONS

In this section we present applications that we are considering for OceanStore. While each of these applications can be constructed in isolation, OceanStore enables them to be developed more easily and completely by providing a single infrastructure for their shared, difficult problems. These problems include consistency, security, privacy, wide-scale data dissemination, dynamic optimization, durable storage, and disconnected operation. OceanStore solves these problems once, allowing application developers to focus on higher-level concerns.

One obvious class of applications for OceanStore is that of groupware and personal information management tools, such as calendars, email, contact lists, and distributed design tools. These applications are challenging to implement because they must allow for concurrent updates from many people. Further, they require that users see an ever-progressing view of shared information, even when conflicts occur. OceanStore's flexible update mechanism solves many of these problems. It provides ways to merge information and detect conflicts, as well as the infrastructure to disseminate information to all interested parties. Additionally, OceanStore provides ubiquitous access to data so that any device can access the information from anywhere.

Email is a particularly interesting groupware target for OceanStore. Although email applications appear mundane on the surface, their implementations are difficult because the obvious solution of filtering all messages through a single email server does not scale well, and distributed solutions have complicated internal consistency issues. For example, an email inbox may be simultaneously written by numerous different users while being read by a single user. Further, some operations, such as message move operations, must occur atomically even in the face of concurrent access from several clients to avoid data loss. In addition, email requires privacy and security by its very nature. OceanStore alleviates the need for clients to implement their own locking and security mechanisms, while enabling powerful features such as nomadic email collections and disconnected operation. Introspection permits a user's email to migrate closer to his client, reducing the round trip time to fetch messages from a remote server. OceanStore enables disconnected operation through its optimistic concurrency model—users can operate on locally cached email even when disconnected from the network; modifications are automatically disseminated upon reconnection.

In addition to groupware applications, OceanStore can be used to create very large digital libraries and repositories for scientific data. Both of these applications require massive quantities of storage, which in turn require complicated management. OceanStore provides a common mechanism for storing and managing these large data collections. It replicates data for durability and availability. Its deep archival storage mechanisms permit information to survive in the face of global disaster. Further, OceanStore benefits these applications by providing for seamless migration of data to where it is needed. For example, OceanStore can quickly disseminate vast streams of data from physics laboratories to the researchers around the world who analyze such data.

Finally, OceanStore provides an ideal platform for new streaming applications, such as sensor data aggregation and dissemination. Many have speculated about the utility of data that will emanate from the plethora of small MEMS sensors in the future; OceanStore provides a uniform infrastructure for transporting, filtering, and aggregating the huge volumes of data that will result.

# 4 SYSTEM ARCHITECTURE

In this section, we will describe underlying technologies that support the OceanStore system. We start with basic issues, such as *naming* and *access control*. We proceed with a description of the *data location* mechanism, which must locate objects anywhere in the world. Next, we discuss the OceanStore *update model* and the issues involved with *consistency management* in an untrusted infrastructure. After a brief word on the architecture for *archival storage*, we discuss the OceanStore API as presented to clients. Finally, we provide a description of the role of *introspection* in OceanStore.

## 4.1 Naming

At the lowest level, OceanStore objects are identified by a *globally unique identifier* (GUID), which can be thought of as a pseudo-random, fixed-length bit string. Users of the system, however, will clearly want a more accessible naming facility. To provide a facility that is both decentralized and resistant to attempts by adversaries to "hijack" names that belong to other users, we have adapted the idea of self-certifying path names due to Mazières [35].

An object GUID is the secure hash[3] of the owner's key and some human-readable name. This scheme allows servers to verify an object's owner efficiently, which facilitates access checks and resource accounting[4].

Certain OceanStore objects act as directories, mapping human-readable names to GUIDs. To allow arbitrary directory hierarchies to be built, we allow directories to contain pointers to other directories. A user of the OceanStore can choose several directories as "roots" and secure those directories through external methods, such as a public key authority. Note, however, that such root directories are only roots with respect to the clients that use them; the system as a whole has no one root. This scheme does not solve the problem of generating a secure GUID mapping, but rather reduces it to a problem of secure key lookup. We address this problem using the locally linked name spaces from the SDSI framework [1, 42].

Note that GUIDs identify a number of other OceanStore entities such as servers and archival fragments. The GUID for a server is a secure hash of its public key; the GUID for an archival fragment is a secure hash over the data it holds. As described in Section 4.3, entities in the OceanStore may be addressed directly by their GUID.

## 4.2 Access control

OceanStore supports two primitive types of access control, namely *reader* restriction and *writer* restriction. More complicated access control policies, such as working groups, are constructed from these two.

**Restricting readers:** To prevent unauthorized reads, we encrypt all data in the system that is not completely public and distribute the encryption key to those users with read permission. To revoke read permission, the owner must request that replicas be deleted or re-encrypted with the new key. A recently-revoked reader is able

[3] Our prototype system uses SHA-1 [37] for its secure hash.

[4] Note that each user might have more than one public key. They might also choose different public keys for private objects, public objects, and objects shared with various groups.

to read old data from cached copies or from misbehaving servers that fail to delete or re-key; however, this problem is not unique to OceanStore. Even in a conventional system, there is no way to force a reader to forget what has been read.

**Restricting writers:** To prevent unauthorized writes, we require that all writes be signed so that well-behaved servers and clients can verify them against an access control list (ACL). The owner of an object can securely choose the ACL *x* for an object *foo* by providing a signed certificate that translates to "Owner says use ACL *x* for object *foo*". The specified ACL may be another object or a value indicating a common default. An ACL entry extending privileges must describe the privilege granted and the signing key, but not the explicit identity, of the privileged users. We make such entries publicly readable so that servers can check whether a write is allowed. We plan to adopt ideas from systems such as Taos and PolicyMaker to allow users to express and reason formally about a wide range of possible policies [52, 6].

Note the asymmetry that has been introduced by encrypted data: reads are restricted at clients via key distribution, while writes are restricted at servers by ignoring unauthorized updates.

## 4.3 Data Location and Routing

Entities in the OceanStore are free to reside on *any* of the OceanStore servers. This freedom provides maximum flexibility in selecting policies for replication, availability, caching, and migration. Unfortunately, it also complicates the process of locating and interacting with these entities. Rather than restricting the placement of data to aid in the location process, OceanStore tackles the problem of data location head-on. The paradigm is that of query routing, in which the network takes an active role in routing messages to objects.

### *4.3.1 Distributed Routing in OceanStore*

Every addressable entity in the OceanStore (*e*.g. floating replica, archival fragment, or client) is identified by one or more GUIDs. Entities that are functionally equivalent, such as different replicas for the same object, are identified by the same GUID. Clients interact with these entities with a series of protocol messages, as described in subsequent sections. To support location-independent addressing, OceanStore messages are labeled with a destination GUID, a random number, and a small predicate. The destination IP address does not appear in these messages. The role of the OceanStore routing layer is to route messages directly to the closest node that matches the predicate and has the desired GUID.

In order perform this routing process, the OceanStore networking layer consults a distributed, fault-tolerant data structure that explicitly tracks the location of all objects. Routing is thus a two phase process. Messages begin by routing from node to node along the distributed data structure until a destination is discovered. At that point, they route directly to the destination. It is important to note that the OceanStore routing layer does not supplant IP routing, but rather provides additional functionality on top of IP.

There are many advantages to combining data location and routing in this way. First and foremost, the task of routing a particular message is handled by the aggregate resources of many different nodes. By exploiting multiple routing paths to the destination, this serves to limit the power of compromised nodes to deny service to a client. Second, messages route directly to their destination, avoiding the multiple round-trips that a separate data location and routing process would incur. Finally, the underlying infrastructure has more up-to-date information about the current location of entities than the clients. Consequently, the combination of location and routing permits communication with "the closest" entity, rather than an entity that the client might have heard of in the past. If replicas move around, only the network, not the users of the data, needs to know.

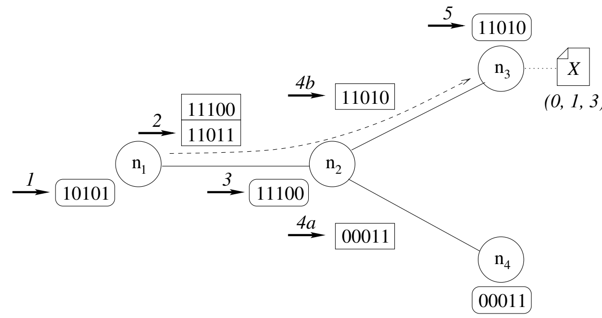


**Figure 2: *The probabilistic query process.* The replica at $n_1$ is looking for object $X$, whose GUID hashes to bits 0, 1, and 3. (1) The local Bloom filter for $n_1$ (rounded box) shows that it does not have the object, but (2) its neighbor filter (unrounded box) for $n_2$ indicates that $n_2$ might be an intermediate node en route to the object. The query moves to $n_2$, (3) whose Bloom filter indicates that it does not have the document locally, (4a) that its neighbor $n_4$ doesn't have it either, but (4b) that its neighbor $n_3$ might. The query is forwarded to $n_3$, (5) which verifies that it has the object.**

The mechanism for routing is a two-tiered approach featuring a fast, probabilistic algorithm backed up by a slower, reliable hierarchical method. The justification for this two-level hierarchy is that entities that are accessed frequently are likely to reside close to where they are being used; mechanisms to ensure this locality are described in Section 4.7. Thus, the probabilistic algorithm routes to entities rapidly if they are in the local vicinity. If this attempt fails, a large-scale hierarchical data structure in the style of Plaxton et. al. [40] locates entities that cannot be found locally. We will describe these two techniques in the following sections.

### *4.3.2 Attenuated Bloom Filters*

The probabilistic algorithm is fully distributed and uses a constant amount of storage per server. It is based on the idea of hill-climbing; if a query cannot be satisfied by a server, local information is used to route the query to a likely neighbor. A modified version of a Bloom filter [7]—called an *attenuated* Bloom filter—is used to implement this potential function.

An attenuated Bloom filter of depth $D$ can be viewed as an array of $D$ normal Bloom filters. In the context of our algorithm, the first Bloom filter is a record of the objects contained locally on the current node. The $i$th Bloom filter is the union of all of the Bloom filters for all of the nodes a distance $i$ through any path from the current node. An attenuated Bloom filter is stored for each directed edge in the network. A query is routed along the edge whose filter indicates the presence of the object at the smallest distance. This process is illustrated in Figure 2. Our current metric of distance is hop-count, but in the future we hope to include a more precise measure corresponding roughly to latency. Also, "reliability factors" can be applied locally to increase the distance to nodes that have abused the protocol in the past, automatically routing around certain classes of attacks.

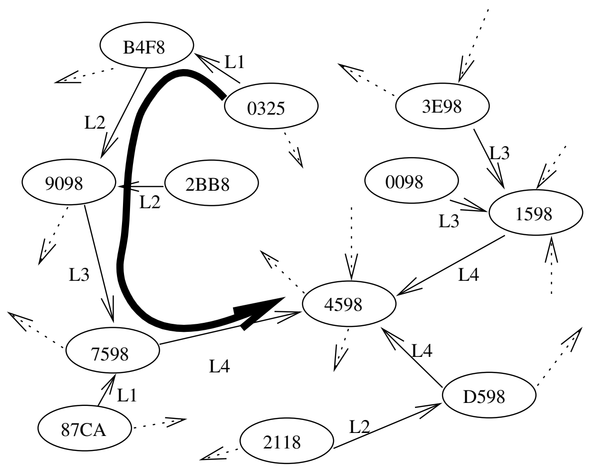


**Figure 3: A portion of the global mesh, rooted at node 4598. Paths from any node to the root of any tree can be traversed by resolving the root's ID one digit at a time; the bold arrow shows a route from node 0325 to node 4598. Data location uses this structure. Note that most object searches do not travel all the way to the root (see text).**

#### *4.3.3 The Global Algorithm: Wide-scale Distributed Data Location*

The global algorithm for the OceanStore is a variation on Plaxton et. al.'s randomized hierarchical distributed data structure [40], which embeds multiple random trees in the network. Although OceanStore uses a highly-redundant version of this data structure, it is instructive to understand the basic Plaxton scheme. In that scheme, every server in the system is assigned a random (and unique) node-ID. These node-IDs are then used to construct a mesh of neighbor links, as shown in Figure 3. In this figure, each link is labeled with a level number that denotes the stage of routing that uses this link. In the example, the links are constructed by taking each node-ID and dividing it into chunks of four bits. The N$^{th}$ level neighbor-links for some Node X point at the 16 *closest neighbors*[5] whose node-IDs match the lowest N-1 nibbles of Node X's ID and who have different combinations of the N$^{th}$ nibble; one of these links is always a loopback link. If a link cannot be constructed because no such node meets the proper constraints, then the scheme chooses the node that matches the constraints as closely as possible. This process is repeated for all nodes and levels within a node.

The key observation to make from Figure 3 is that the links form a series of random embedded trees, with each node as the root of one of these trees. As a result, the neighbor links can be used to route from anywhere to a given node, simply by resolving the node's address one link at a time—first a level-one link, then a level-two link, etc. To use this structure for data location, we map each object to a single node whose node-ID matches the object's GUID in the most bits (starting from the least significant); call this node the object's *root*. If information about the GUID (such as its location) were stored at its root, then anyone could find this information simply by following neighbor links until they reached the root node for the GUID. As described, this scheme has nice load distribution properties, since GUIDs become randomly mapped throughout the infrastructure.

This random distribution would appear to reduce locality; however, the Plaxton scheme achieves locality as follows: when a replica is placed somewhere in the system, its location is "published" to the routing infrastructure. The publishing process works its way to the object's root and deposits a pointer at every hop along the way. This process requires $O(\log n)$ hops, where $n$ is the number of servers in the world. When someone searches for information, they climb the tree until they run into a pointer, after which they route directly to the object. In [40], the authors show that the average distance traveled is proportional to the distance between the source of the query and the closest replica that satisfies this query.

**Achieving Fault Tolerance:** The basic scheme described above is sensitive to a number of different failures. First, each object has a single root, which becomes a single point of failure, the potential subject of denial of service attacks, and an availability problem. OceanStore addresses this weakness in a simple way: it hashes each GUID with a small number of different salt values. The result maps to several different root nodes, thus gaining redundancy and simultaneously making it difficult to target a single node with a denial of service attack against a range of GUIDs.

A second problem with the above scheme is sensitivity to corruption in the links and pointers. An important observation, however, is that the above structure has sufficient redundancy to tolerate small amounts of corruption. Bad links can be immediately detected, and routing can be continued by jumping to a random neighbor node[6]. To increase this redundancy, the OceanStore location structure supplements the basic links of the above scheme with additional neighbor links. Further, the infrastructure continually monitors and repairs neighbor links (a form of *introspection*—see Section 4.7), and servers slowly repeat the publishing process to repair pointers.

**The Advantages of Distributed Information:** The advantages of a Plaxton-like data structure in the OceanStore are many. First, it is a highly redundant and fault-tolerant structure that spreads data location load evenly while finding local objects quickly. The combination of the probabilistic and global algorithms should comfortably scale to millions of servers. Second, the aggregate information contained in this data structure is sufficient to recognize which servers are down and to identify data that must be reconstructed when a server is permanently removed. This feature is important for maintaining a minimum level of redundancy for the deep archival storage. Finally, the Plaxton links form a natural substrate on which to perform network functions such as admission control and multicast.

**Achieving Maintenance-Free Operation:** While existing work on Plaxton-like data structures did not include algorithms for online creation and maintenance of the global mesh, we have produced recursive node insertion and removal algorithms. These make use of the redundant neighbor links mentioned above. Further, we have generalized our publication algorithm to support replicated roots, which remove single-points of failure in data location. Finally, we have optimized failure modes by using soft-state beacons to detect faults more quickly, time-to-live fields to react better to routing updates, and a second-chance algorithm to minimize the cost of recovering lost nodes. This information is coupled with continuous repair mechanisms that recognize when

[5]"Closest" means with respect to the underlying IP routing infrastructure. Roughly speaking, the measurement metric is the time to route via IP.

[6]Each tree spans every node, hence any node should be able to reach the root.

servers have been down for a long time and need to have their data reconstructed[7]. The practical implication of this work is that the OceanStore infrastructure as a whole automatically adapts to the presence or absence of particular servers without human intervention, greatly reducing the cost of management.

## 4.4 Update Model

Several of the applications described in Section 3 exhibit a high degree of write sharing. To allow for concurrent updates while avoiding many of the problems inherent with wide-area locking, OceanStore employs an update model based on *conflict resolution*. Conflict resolution was introduced in the Bayou system [13] and supports a range of consistency semantics—up to and including ACID semantics. Additionally, conflict resolution reduces the number of aborts normally seen in detection-based schemes such as optimistic concurrency control [29].

Although flexible, conflict resolution requires the ability to perform server-side computations on data. In an untrusted infrastructure, replicas have access only to ciphertext, and no one server is trusted to perform commits. Both of these issues complicate the update architecture. However, the current OceanStore design is able to handle many types of conflict resolution directly on encrypted data. The following paragraphs describe the issues involved and our progress towards solving them.

### 4.4.1 Update Format and Semantics

Changes to data objects within OceanStore are made by client-generated *updates*, which are lists of predicates associated with actions. The semantics of an update are as follows: to apply an update against a data object, a replica evaluates each of the update's predicates in order. If any of the predicates evaluates to true, the actions associated with the earliest true predicate are atomically applied to the data object, and the update is said to *commit*. Otherwise, no changes are applied, and the update is said to *abort*. The update itself is logged regardless of whether it commits or aborts.

Note that OceanStore update semantics are similar to those of the Bayou system, except that we have eliminated the merge procedure used there, since arbitrary computations and manipulations on ciphertext are still intractable. Nevertheless, we preserve the key functionality of their model, which they found to be expressive enough for a number of sample applications including a group calendar, a shared bibliographic database, and a mail application [14]. Furthermore, the model can be applied to other useful applications. For instance, Coda [26] provided specific merge procedures for conflicting updates of directories; this type of conflict resolution is easily supported under our model. Slight extensions to the model can support Lotus Notes-style conflict resolution, where unresolvable conflicts result in a branch in the object's version stream [25]. Finally, the model can be used to provide ACID semantics: the first predicate is made to check the read set of a transaction, the corresponding action applies the write set, and there are no other predicate-action pairs.

### 4.4.2 Extending the Model to Work over Ciphertext

OceanStore replicas are not trusted with unencrypted information. This complicates updates by restricting the set of predicates that replicas can compute and the set of actions they are able to apply. However, the following predicates are currently possible: *compare-version*, *compare-size*, *compare-block*, and *search*. The first two predicates are trivial since they are over the unencrypted meta-data of the object. The *compare-block* operation is easy if the encryption technology is a position-dependent block cipher: the client simply computes a hash of the encrypted block and submits it along with the block number for comparison. Perhaps the most impressive of these predicates is *search*, which can be performed directly on ciphertext [47]; this operation reveals only that a search was performed along with the boolean result. The cleartext of the search string is not revealed, nor can the server initiate new searches on its own.

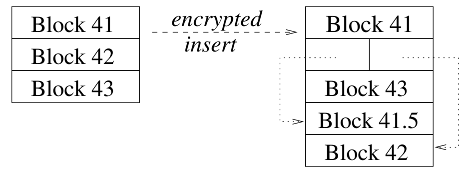


**Figure 4: *Block insertion on ciphertext.* The client wishes to insert block 41.5, so she appends it and block 42 to the object, then replaces the old block 42 with a block pointing to the two appended blocks. The server learns nothing about the contents of any of the blocks.**

In addition to these predicates, the following operations can be applied to ciphertext: *replace-block*, *insert-block*, *delete-block*, and *append*. Again assuming a position-dependent block cipher, the *replace-block* and *append* operations are simple for the same reasons as *compare-block*.

The last two operations, *insert-block* and *delete-block*, can be performed by grouping blocks of the object into two sets, index blocks and data blocks, where index blocks contain pointers to other blocks elsewhere in the object. To insert, one replaces the block at the insertion point with a new block that points to the old block and the inserted block, both of which are appended to the object. This scheme is illustrated in Figure 4. To delete, one replaces the block in question with an empty pointer block. Note that this scheme leaks a small amount of information and thus might be susceptible to compromise by a traffic-analysis attack; users uncomfortable with this leakage can simply append encrypted log records to an object and rely on powerful clients to occasionally generate and re-encrypt the object in whole from the logs.

The schemes presented in this section clearly impact the format of objects. However, these schemes are the subject of ongoing research; more flexible techniques will doubtless follow.

### 4.4.3 Serializing Updates in an Untrusted Infrastructure

The process of conflict resolution starts with a series of updates, chooses a total order among them, then applies them atomically in that order. The easiest way to compute this order is to require that all updates pass through a *master* replica. Unfortunately, trusting any one replica to perform this task is incompatible with the untrusted infrastructure assumption on which OceanStore is built. Thus, we replace this master replica with a *primary tier* of replicas. These replicas cooperate with one another in a Byzantine agreement protocol [30] to choose the final commit order for updates[8]. A secondary tier of replicas communicates among themselves and the primary tier via an enhanced *epidemic* algorithm, as in Bayou.

The decision to use two classes of floating replicas is motivated by several considerations. First, all known protocols that are toler-

[7] Note that the read-only nature of most of the information in the OceanStore makes this reconstruction particularly easy; see Section 4.5.

[8] A Byzantine agreement protocol is one in which we assume that no more than $m$ of the total $n = 3m + 1$ replicas are faulty.

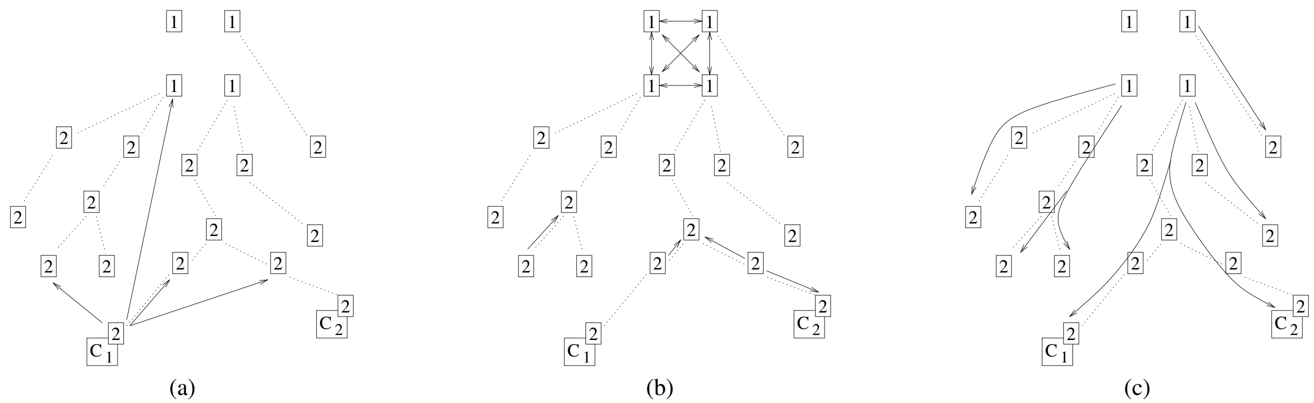


**Figure 5:** ***The path of an update.*** **(a) After generating an update, a client sends it directly to the object's primary tier, as well as to several other random replicas for that object. (b) While the primary tier performs a Byzantine agreement protocol to commit the update, the secondary replicas propagate the update among themselves epidemically. (c) Once the primary tier has finished its agreement protocol, the result of the update is multicast down the dissemination tree to all of the secondary replicas.**

ant to arbitrary replica failures are too communication-intensive to be used by more than a handful of replicas. The primary tier thus consists of a small number of replicas located in high-bandwidth, high-connectivity regions of the network[9]. To allow for later, offline verification by a party who did not participate in the protocol, we are exploring the use of *proactive* signature techniques [4] to certify the result of the serialization process. We hope to extend the protocol in [10] to use such techniques.

Some applications may gain performance or availability by requiring a lesser degree of consistency than ACID semantics. These applications motivate the secondary tier of replicas in OceanStore. Secondary replicas do not participate in the serialization protocol, may contain incomplete copies of an object's data, and can be more numerous than primary replicas. They are organized into one or more application-level multicast trees, called *dissemination trees*, that serve as conduits of information between the primary tier and secondary tier. Among other things, the dissemination trees *push* a stream of committed updates to the secondary replicas, and they serve as communication paths along which secondary replicas *pull* missing information from parents and primary replicas. This architecture permits dissemination trees to transform *updates* into *invalidations* as they progress downward; such a transformation is exploited at the leaves of the network where bandwidth is limited.

Secondary replicas contain both tentative[10] and committed data. They employ an epidemic-style communication pattern to quickly spread tentative commits among themselves and to pick a tentative serialization order. To increase the chances that this tentative order will match the final ordering chosen by the primary replicas, clients optimistically timestamp their updates. Secondary replicas order tentative updates in timestamp order, and the primary tier uses these same timestamps to guide its ordering decisions. Since the serialization decisions of the secondary tier are tentative, they may be safely decided by untrusted replicas; applications requiring stronger consistency guarantees must simply wait for their updates to reach the primary tier.

### 4.4.4 A Direct Path to Clients and Archival Storage

The full path of an update is shown in Figure 5. Note that this path is optimized for low latency and high throughput. Under ideal circumstances, updates flow directly from the client to the primary tier of servers, where they are serialized and then multicast to the secondary servers. All of the messages shown here are addressed through GUIDs, as described in Section 4.3. Consequently, the update protocol operates entirely without reference to the physical location of replicas.

One important aspect of OceanStore that differs from existing systems is the fact that the archival mechanisms are tightly coupled with update activity. After choosing a final order for updates, the inner tier of servers signs the result and sends it through the dissemination tree. At the same time, these servers generate encoded, archival fragments and distribute them widely. Consequently, updates are made extremely durable as a direct side-effect of the commitment process. Section 4.5 discusses archival storage in more detail.

### 4.4.5 Efficiency of the Consistency Protocol

There are two main points of interest when considering the efficiency of the consistency protocol: the amount of network bandwidth the protocol demands, and the latency between when an update is created and when the client receives notification that it has committed or aborted. Assuming that a Byzantine agreement protocol like that in [10] is used, the total cost an update in bytes sent across the network, $b$, is given by the equation:

$$b = c_1 n^2 + (u + c_2)n + c_3$$

where $u$ is the size of the update, $n$ is the number of replicas in the primary tier, and $c_1$, $c_2$, and $c_3$ are the sizes of small protocol messages. While this equation appears to be dominated by the $n^2$ term, the constant $c_1$ is quite small, on the order of 100 bytes. Thus for sufficiently small $n$ and large updates, the equation is dominated by the $n$ term. Since there are $n$ replicas, the minimum amount of bytes that must be transfered to keep all replicas up to date is $un$.

Figure 6 shows the cost of an update, normalized to this minimum amount, as a function of update size. Note that for $m = 4$ and $n = 13$, the normalized cost approaches 1 for update sizes around 100k bytes, but it approaches 2 at update sizes of only around 4k bytes.[11] Thus for updates of 4k bytes or more, our system uses less than double the minimum amount of network bandwidth necessary to keep all the replicas in the primary tier up to date.

[9] The choice of which replicas to include in the primary tier is left to the client's responsible party, which must ensure that its chosen group satisfies the Byzantine assumption mentioned above.

[10] Tentative data is data that the primary replicas have not yet committed.

[11] Recall that $m$ is the number of faulty replicas tolerated by the Byzantine agreement protocol.

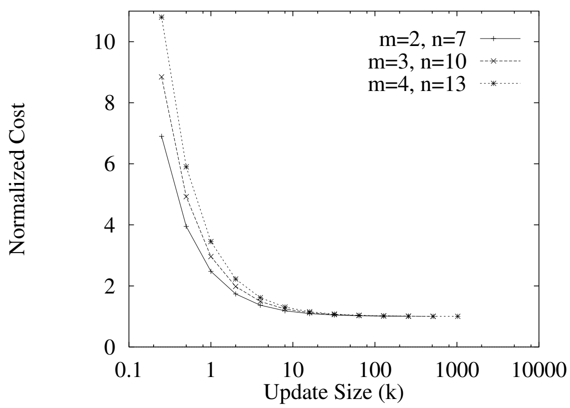


**Figure 6: The cost of an update in bytes sent across the network, normalized to the minimum cost needed to send the update to each of the replicas.**

Unfortunately, latency estimates for the consistency protocol are more difficult to come by without a functioning prototype. For this reason, let us suffice it to say that there are six phases of messages in the protocol we have described. Assuming latency of messages over the wide area dominates computation time and that each message takes 100ms, we have an approximate latency per update of less than a second. We believe this latency is reasonable, but we will need to complete our prototype system before we can verify the accuracy of this rough estimate.

## 4.5 Deep Archival Storage

The archival mechanism of OceanStore employs *erasure codes*, such as interleaved Read-Solomon codes [39] and Tornado codes [32]. Erasure coding is a process that treats input data as a series of fragments (say $n$) and transforms these fragments into a greater number of fragments (say $2n$ or $4n$). As mentioned in Section 4.4, the fragments are generated in parallel by the inner tier of servers during the commit process. The essential property of the resulting code is that *any* $n$ of the coded fragments are sufficient to construct the original data[12].

Assuming that we spread coded fragments widely, it is very unlikely that enough servers will be down to prevent the recovery of data. We call this argument *deep archival storage*. A simple example will help illustrate this assertion. Assuming uncorrelated faults among machines, one can calculate the reliability at a given instant of time according to the following formula:

$$P = \sum_{i=0}^{r_f} \frac{\binom{m}{i}\binom{n-m}{f-i}}{\binom{n}{f}}$$

where $P$ is the probability that a document is available, $n$ is the number of machines, $m$ is the number of currently unavailable machines, $f$ is the number of fragments per document, and $r_f$ is the maximum number of unavailable fragments that still allows the document to be retrieved. For instance, with a million machines, ten percent of which are currently down, simple replication without erasure codes provides only two nines (0.99) of reliability. A $1/2$-rate erasure coding of a document into 16 fragments gives the document over five nines of reliability (0.999994), yet consumes the same amount of storage. With 32 fragments, the reliability increases by another factor of 4000, supporting the assertion that *fragmentation increases reliability*. This is a consequence of the law of large numbers.

To preserve the erasure nature of the fragments (meaning that a fragment is either retrieved correctly and completely, or not at all), we use a hierarchical hashing method to verify each fragment. We generate a hash over each fragment, and recursively hash over the concatenation of pairs of hashes to form a binary tree. Each fragment is stored along with the hashes neighboring its path to the root. When it is retrieved, the requesting machine may recalculate the hashes along that path. We can use the top-most hash as the GUID to the immutable archival object, making every fragment in the archive completely self-verifying.

For the user, we provide a naming syntax which explicitly incorporates version numbers. Such names can be included in other documents as a form of permanent hyper-link. In addition, interfaces will exist to examine modification history and to set versioning policies [44]. Although in principle every version of every object is archived, clients can choose to produce versions less frequently. Archival copies are also produced when objects are idle for a long time or before objects become inactive. When generating archival fragments, the floating replicas of an object participate together: they each generate a disjoint subset of the fragments and disseminate them into the infrastructure.

To maximize the survivability of archival copies, we identify and rank administrative domains by their reliability and trustworthiness. We avoid dispersing all of our fragments to locations that have a high correlated probability of failure. Further, the number of fragments (and hence the durability of information) is determined on a per-object basis. OceanStore contains processes that slowly sweep through all existing archival data, repairing or increasing the level of replication to further increase durability.

To reconstruct archival copies, OceanStore sends out a request keyed off the GUID of the archival versions. Note that we can make use of excess capacity to insulate ourselves from slow servers by requesting more fragments than we absolutely need and reconstructing the data as soon as we have enough fragments. As the request propagates up the location tree (Section 4.3), fragments are discovered and sent to the requester. This search has nice locality properties since closer fragments tend to be discovered first.

## 4.6 The OceanStore API

OceanStore draws much strength from its global scale, wide distribution, epidemic propagation method, and flexible update policy. The system as a whole can have rather complicated behavior. However, the OceanStore application programming interface (API) enables application writers to understand their interaction with the system.

This base API provides full access to OceanStore functionality in terms of sessions, session guarantees, updates, and callbacks. A session is a sequence of reads and writes to potentially different objects that are related to one another through session guarantees. Guarantees define the level of consistency seen by accesses through a session. The API provides mechanisms to develop arbitrarily complex updates in the form described in Section 4.4. The API also provides a callback feature to notify applications of relevant events. An application can register an application-level handler to be invoked at the occurrence of relevant events, such as the commit or abort of an update.

Applications with more basic requirements are supported through facades to the standard API. A facade is an interface to the API that provides a traditional, familiar interface. For example, a transaction facade would provide an abstraction atop the OceanStore API so that the developer could access the system

[12] Tornado codes, which are faster to encode and decode, require slightly more than $n$ fragments to reconstruct the information.

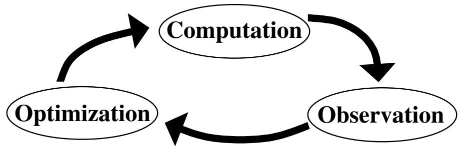


**Figure 7: The Cycle of Introspection**

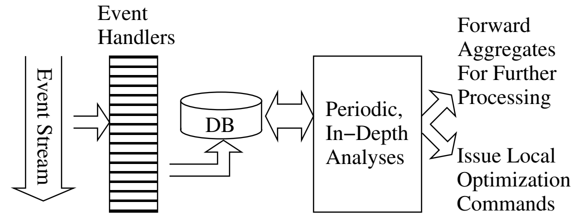


**Figure 8: Fast event handlers summarize and respond to local events. For efficiency, the "database" may be only soft state (see text). Further processing analyzes trends and aggregate information across nodes.**

in terms of traditional transactions. The facade would simplify the application writer's job by ensuring proper session guarantees, reusing standard update templates, and automatically computing read sets and write sets for each update.

Of course, OceanStore is a new system in a world of legacy code, and it would be unreasonable to expect the authors of existing applications to port their work to an as yet undeployed system. Therefore, OceanStore provides a number of legacy facades that implement common APIs, including a Unix file system, a transactional database, and a gateway to the World Wide Web. These interfaces exist as libraries or "plugins" to existing browsers or operating systems. They permit users to access legacy documents while enjoying the ubiquitous and secure access, durability, and performance advantages of OceanStore.

## 4.7 Introspection

As envisioned, OceanStore will consist of millions of servers with varying connectivity, disk capacity, and computational power. Servers and devices will connect, disconnect, and fail sporadically. Server and network load will vary from moment to moment. Manually tuning a system so large and varied is prohibitively complex. Worse, because OceanStore is designed to operate using the utility model, manual tuning would involve cooperation across administrative boundaries.

To address these problems, OceanStore employs *introspection*, an architectural paradigm that mimics adaptation in biological systems. As shown in Figure 7, introspection augments a system's normal operation (*computation*), with *observation* and *optimization*. *Observation modules* monitor the activity of a running system and keep a historical record of system behavior. They also employ sophisticated analyses to extract patterns from these observations. *Optimization modules* use the resulting analysis to adjust or adapt the computation.

OceanStore uses introspective mechanisms throughout the system. Although we have insufficient space to describe each use in detail, we will give a flavor of our techniques below.

### *4.7.1 Architecture*

We have designed a common architecture for introspective systems in OceanStore (see Figure 8). These systems process local events, forwarding summaries up a distributed hierarchy to form approximate global views of the system. Events include any incoming message or noteworthy physical measurement. Our three-point approach provides a framework atop which we are developing specific observation and optimization modules.

The high event rate[13] precludes extensive online processing. Instead, a level of fast event handlers summarizes local events. These summaries are stored in a local database. At the leaves of the hierarchy, this database may reside only in memory; we loosen durability restrictions for local observations in order to attain the necessary event rate.

[13] Each machine initiates and receives roughly as many messages as local area network files systems. In addition, the routing infrastructure requires communication proportional to the logarithm of the size of the network.

We describe all event handlers in a simple domain-specific language. This language includes primitives for operations like averaging and filtering, but explicitly prohibits loops. We expect this model to provide sufficient power, flexibility, and extensibility, while enabling the verification of security and resource consumption restrictions placed on event handlers.

A second level of more powerful algorithms periodically processes the information in the database. This level can perform sophisticated analyses and incorporate historical information, allowing the system to detect and respond to long-term trends.

Finally, after processing and responding to its own events, a third level of each node forwards an appropriate summary of its knowledge to a parent node for further processing on the wider scale. The infrastructure uses the standard OceanStore location mechanism to locate that node, which is identified by its GUID. Conversely, we could distribute the information to remote optimization modules as OceanStore objects that would also be accessed via the standard location mechanism.

### *4.7.2 Uses of Introspection*

We use introspection to manage a number of subsystems in the OceanStore. Below, we will discuss several of these components.

**Cluster Recognition:** Cluster recognition attempts to identify and group closely related files. Each client machine contains an event handler triggered by each data object access. This handler incrementally constructs a graph representing the semantic distance [28] among data objects, which requires only a few operations per access.

Periodically, we run a clustering algorithm that consumes this graph and detects clusters of strongly-related objects. The frequency of this operation adapts to the stability of the input and the available processing resources. The result of the clustering algorithm is forwarded to a global analysis layer that publishes small objects describing established clusters. Like directory listings, these objects help remote optimization modules collocate and prefetch related files.

**Replica Management:** Replica management adjusts the number and location of floating replicas in order to service access requests more efficiently. Event handlers monitor client requests and system load, noting when access to a specific replica exceeds its resource allotment. When access requests overwhelm a replica, it forwards a request for assistance to its parent node. The parent, which tracks locally available resources, can create additional floating replicas on nearby nodes to alleviate load.

Conversely, replica management eliminates floating replicas that have fallen into disuse. Notification of a replica's termination also propagates to parent nodes, which can adjust that object's dissemination tree.

In addition to these short-term decisions, nodes regularly analyze global usage trends, allowing additional optimizations. For example, OceanStore can detect periodic migration of clusters from site to site and prefetch data based on these cycles. Thus users will find their project files and email folder on a local machine during the work day, and waiting for them on their home machines at night.

**Other Uses:** OceanStore uses introspective mechanisms in many other aspects as well. Specifically, introspection improves the manageability and performance of the routing structure, enables construction of efficient update dissemination trees, ensures the availability and durability of archival fragments, identifies unreliable peer organizations, and performs continuous confidence estimation on its own optimizations in order to reduce harmful changes and feedback cycles.

# 5 STATUS

We are currently implementing an OceanStore prototype that we will deploy for testing and evaluation. The system is written in Java with a state machine-based request model for fast I/O [22]. Initially, OceanStore will communicate with applications through a UNIX file system interface and a read-only proxy for the World Wide Web in addition to the native OceanStore API.

We have explored the requirements that our security guarantees place on a storage architecture. Specifically, we have explored differences between enforcing read and write permissions in an untrusted setting, emphasizing the importance of the ability of clients to validate the correctness of any data returned to them. This exploration included not only checking the integrity of the data itself, but also checking that the data requested was the data returned, and that all levels of metadata were protected as strongly as the data itself. A prototype cryptographic file system provided a testbed for specific security mechanisms.

A prototype for the probabilistic data location component has been implemented and verified. Simulation results show that our algorithm finds nearby objects with near-optimal efficiency.

We have implemented prototype archival systems that use both Reed-Solomon and Tornado codes for redundancy encoding. Although only one half of the fragments were required to reconstruct the object, we found that issuing requests for extra fragments proved beneficial due to dropped requests.

We have implemented the introspective prefetching mechanism for a local file system. Testing showed that the method correctly captured high-order correlations, even in the presence of noise. We will combine that mechanism with an optimization module appropriate for the wide-area network.

# 6 RELATED WORK

Distributed systems such as Taos [52] assume untrusted networks and applications, but rely on some trusted computing base. Cryptographic file systems such as Blaze's CFS [5] provide end-to-end secrecy, but include no provisions for sharing data, nor for protecting integrity independently from secrecy. The Secure File System [24] supports sharing with access control lists, but fails to provide independent support for integrity, and trusts a single server to distribute encryption keys. The Farsite project [8] is more similar to OceanStore than these other works, but while it assumes the use of untrusted clients, it does not address a wide-area infrastructure.

SDSI [1] and SPKI [15] address the problem of securely distributing keys and certificates in a decentralized manner. PolicyMaker [6] deals with the description of trust relations. Mazières proposes self-certifying paths to separate key management from system security [35].

Bloom filters [7] are commonly used as compact representations of large sets. The R* distributed database [33] calculates them on demand to implement efficient semijoins. The Summary Cache [16] pushes Bloom filters between cooperating web caches, although their method does not scale well in the number of caches.

Distributing data for performance, availability, or survivability has been studied extensively in both the file systems and database communities. A summary of distributed file systems can be found in [31]. In particular, Bayou [13] and Coda [26] use replication to improve availability at the expense of consistency and introduce specialized conflict resolution procedures. Sprite [36] also uses replication and caching to improve availability and performance, but has a guarantee of consistency that incurs a performance penalty in the face of multiple writers. None of these systems addresses the range of security concerns that OceanStore does, although Bayou examines some problems that occur when replicas are corrupted [48].

Gray et. al. argue against promiscuous replication in [19]. OceanStore differs from the class of systems they describe because it does not bind floating replicas to specific machines, and it does not replicate all objects at each server.

OceanStore's second tier of floating replicas are similar to transactional caches; in the taxonomy of [17] our algorithm is detection-based and performs its validity checks at commit time. In contrast to similar systems, our merge predicates should decrease the number of transactions aborted due to out-of-date caches.

Many previous projects have explored feedback-driven adaptation in extensible operating systems [45], databases [11], file systems [34], global operating systems [9], and storage devices [51]. Although these projects employ differing techniques and terminology, each could be analyzed with respect to the *introspective* model.

The Seer project formulated the concept of semantic distance [28] and collects clusters of related files for automated hoarding. Others have used file system observation to drive automatic prefetching [20, 27].

Introspective replica management for web content was examined in AT&T's Radar project [41], which considers read-only data in a trusted infrastructure. The Mariposa project [46] addresses inter-domain replication with an economic model. Others optimize communication cost when selecting a new location for replica placement [2] within a single administrative domain.

Similar to OceanStore, the Intermemory project [18] uses Cauchy Reed-Solomon Codes to archive wide scale durability. We anticipate that our combination of active and archival object forms will allow greater update performance while retaining Intermemory's survivability benefits.

# 7 CONCLUSION

The rise of ubiquitous computing has spawned an urgent need for persistent information. In this paper we presented OceanStore, a utility infrastructure designed to span the globe and provide secure, highly available access to persistent objects.

Several properties distinguish OceanStore from other systems: the *utility model*, the *untrusted infrastructure*, support for truly *nomadic data*, and use of *introspection* to enhance performance and maintainability. A utility model makes the notion of a global system possible, but introduces the possibility of untrustworthy servers in the system. To this end, we assume that servers may be run by adversaries and cannot be trusted with cleartext; as a result, server-side operations such as conflict-resolution must be performed directly on encrypted information. Nomadic data permits a wide range of optimizations for access to information by bringing it "close" to where it is needed, and enables rapid response to regional outages and denial-of-service attacks. These optimizations are assisted by introspection, the continuous online collection and analysis of access patterns.

OceanStore is under construction. This paper presented many of the design elements and algorithms of OceanStore; several have been implemented. Hopefully, we have convinced the reader that an infrastructure such as OceanStore *is* possible to construct; that it is desirable should be obvious.

# 8 ACKNOWLEDGEMENTS

We would like to thank the following people who have been instrumental in helping us to refine our thoughts about OceanStore (in alphabetical order): William Bolosky, Michael Franklin, Jim Gray, James Hamilton, Joseph Hellerstein, Anthony Joseph, Josh MacDonald, David Patterson, Satish Rao, Dawn Song, Bill Tetzlaff, Doug Tygar, Steve Weis, and Richard Wheeler.

In addition, we would like to acknowledge the enthusiastic support of our DARPA program manager, Jean Scholtz, and industrial funding from EMC and IBM.

# 9 REFERENCES

[1] M. Abadi. On SDSI's linked local name spaces. In *Proc. of IEEE CSFW*, 1997.

[2] S. Acharya and S. B. Zdonik. An efficient scheme for dynamic data replication. Technical Report CS-93-43, Department of Computer Science, Brown University, 1993.

[3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. In *Proc. of ACM SOSP*, Dec. 1995.

[4] B. Barak, A. Herzberg, D. Naor, and E. Shai. The proactive security toolkit and applications. In *Proc. of ACM CCS Conf.*, pages 18–27, Nov. 1999.

[5] M. Blaze. A cryptographic file system for UNIX. In *Proc. of ACM CCS Conf.*, Nov. 1993.

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of IEEE SRSP*, May 1996.

[7] B. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13(7), pages 422–426, July 1970.

[8] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. of Sigmetrics*, June 2000.

[9] W. Bolosky, R. Draves, R. Fitzgerald, C. Fraser, M. Jones, T. Knoblock, and R. Rashid. Operating systems directions for the next millennium. In *Proc. of HOTOS Conf.*, May 1997.

[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of USENIX Symp. on OSDI*, 1999.

[11] S. Chaudhuri and V. Narasayya. AutoAdmin "what-if" index analysis utility. In *Proc. of ACM SIGMOD Conf.*, pages 367–378, June 1998.

[12] M. Dahlin, T. Anderson, D. Patterson, and R. Wang. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of USENIX Symp. on OSDI*, Nov. 1994.

[13] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of IEEE Workshop on Mobile Computing Systems & Applications*, Dec. 1994.

[14] W. Edwards, E. Mynatt, K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *Proc. of ACM Symp. on User Interface Software & Technology*, pages 119–128, 1997.

[15] C. Ellison, B. Frantz, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, 1999.

[16] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. of ACM SIGCOMM Conf.*, pages 254–265, Sept. 1998.

[17] M. Franklin, M. Carey, and M. Livny. Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems*, 22(3):315–363, Sept. 1997.

[18] A. Goldberg and P. Yianilos. Towards an archival intermemory. In *Proc. of IEEE ADL*, pages 147–156, Apr. 1998.

[19] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of ACM SIGMOD Conf.*, volume 25, 2, pages 173–182, June 1996.

[20] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proc. of USENIX Summer Technical Conf.*, June 1994.

[21] E. Hagersten, A. Landin, and S. Haridi. DDM — A Cache-only Memory Architecture. *IEEE Computer*, Sept. 1992.

[22] J. Hill, R. Szewczyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *Proc. of ASPLOS*, Nov. 2000.

[23] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[24] J. Hughes, C. Feist, H. S, M. O'Keefe, and D. Corcoran. A universal access, smart-card-based secure file system. In *Proc. of the Atlanta Linux Showcase*, Oct. 1999.

[25] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *Proc. of ACM CSCW Conf.*, Sept. 1988.

[26] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[27] T. Kroeger and D. Long. Predicting file-system actions from prior events. In *Proc. of USENIX Winter Technical Conf.*, pages 319–328, Jan. 1996.

[28] G. Kuenning. The design of the seer predictive caching system. In *Proc. of IEEE Workshop on Mobile Computing Systems & Applications*, Dec. 1994.

[29] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[30] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, 1982.

[31] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4):321–375, Dec. 1990.

[32] M. Luby, M. Mitzenmacher, M. Shokrollahi, D. Spielman, and V. Stemann. Analysis of low density codes and improved designs using irregular graphs. In *Proc. of ACM STOC*, May 1998.

[33] L. Mackert and G. Lohman. R* optimizer validation and performance for distributed queries. In *Proc. of Intl. Conf. on VLDB*, Aug. 1986.

[34] J. Matthews, D. Roselli, A. Costello, R. Wang, and T. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. of ACM SOSP*, Oct. 1997.

[35] D. Mazières, M. Kaminsky, F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. of ACM SOSP*, 1999.

[36] M. Nelson, B. Welch, and J. Ousterhout. Caching in the sprite network file system. *IEEE/ACM Transactions on Networking*, 6(1):134–154, Feb. 1988.

[37] NIST. FIPS 186 digital signature standard. May 1994.

[38] D. Norman. *The Invisible Computer*, pages 62–63. MIT Press, Cambridge, MA, 1999.

[39] J. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software Practice and Experience*, 27(9):995–1012, Sept. 1997.

[40] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of ACM SPAA*, pages 311–320, Newport, Rhode Island, June 1997.

[41] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *Proc. of IEEE ICDCS*, pages 101–113, June 1999.

[42] R. Rivest and B. Lampson. SDSI—A simple distributed security infrastructure. Manuscript, 1996.

[43] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. of USENIX Summer Technical Conf.*, June 1985.

[44] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. of ACM SOSP*, Dec. 1999.

[45] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proc. of HOTOS Conf.*, pages 124–129, May 1997.

[46] J. Sidell, P. Aoki, S. Barr, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proc. of IEEE ICDE*, pages 485–495, Feb. 1996.

[47] D. Song, D. Wagner, and A. Perrig. Search on encrypted data. To be published in *Proc. of IEEE SRSP*, May 2000.

[48] M. Spreitzer, M. Theimer, K. Petersen, A. Demers, and D. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *Proc. of ACM/IEEE MobiCom Conf.*, pages 234–240, Sept. 1997.

[49] M. Stonebraker. The design of the Postgres storage system. In *Proc. of Intl. Conf. on VLDB*, Sept. 1987.

[50] M. Weiser. The computer for the twenty-first centry. *Scientific American*, Sept. 1991.

[51] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, pages 108–136, Feb. 1996.

[52] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. In *Proc. of ACM SOSP*, pages 256–269, Dec. 1993.