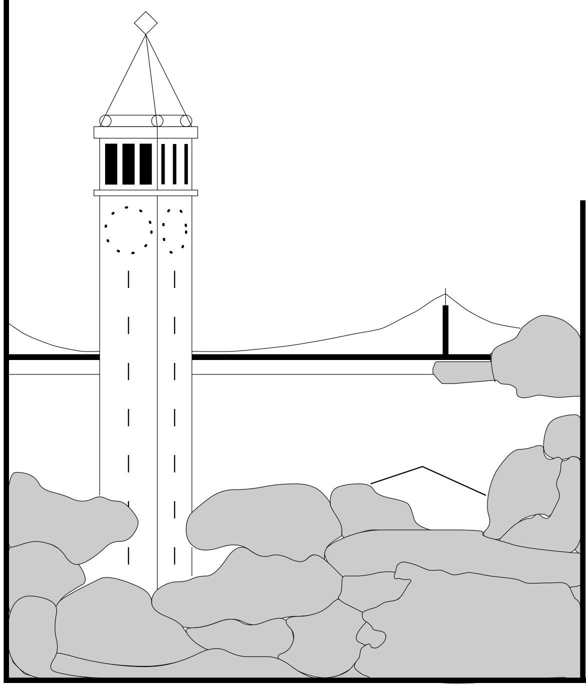
OceanStore: An Extremely Wide–Area Storage System

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



Report No. UCB/CSD-00-1102

Computer Science Division (EECS)
University of California
Berkeley, California 94720

OceanStore: An Extremely Wide–Area Storage System*

Technical Report UCB/CSD-00-1102

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

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. Finally, 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

The computer revolution has occurred. 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. Today, globe-trotting executives routinely access information on their laptops, pagers, and cell-phones; small businesses and individuals have computers that rival the mainframes of a decade ago; cars contain sophisticated computing technology to control their timing. 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, houses, and chairs. Before such a revolution can occur, however, computing devices must become so reliable and resilient that they are completely transparent to the user [58].

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. An embedded component can be rebooted or replaced without losing vital configuration information. Further, the loss or destruction of a device does not lead to lost data. Today, the loss of a laptop is a traumatic event precisely because of the irreplaceable data and hours of configuration time that is lost with it. Note that a uniform infrastructure for persistent information can also provide a framework for transparent consistency; this is important since users will access the same information from many different devices, sometimes simultaneously. Today, such sharing is often laboriously accomplished via 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*. This means that 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. Further, although bandwidth in the core of the Internet has been doubling at a incredible rate, *latency* has not been improving as quickly. Also, 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

*To appear in ASPLOS-2000

memory multiprocessors [28, 49].

1.1 OceanStore: a True Data Utility

As a rough estimation, we might imagine providing service to roughly 10^{10} users each with at least 10,000 files. This means that OceanStore must support over 10^{14} files. As mentioned above, network connectivity is growing at a rapid pace, as is the total amount of data storage and server hardware. Unfortunately, despite the level of *physical* connectivity enjoyed by Internet devices, most of these devices are still disconnected at the *protocol* level. At most, small subsets of devices (owned by individual companies) serve as oases of connectivity to provide specialized services. The great opportunity for reliability, availability, and scalability afforded by millions or billions of devices is lost.

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. Actual 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 amongst 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 global utility.

Ideally, a user would entrust all of his or her data to the 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 media to store it on. 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 which 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 [41]. Among other things, only clients can be trusted with cleartext and all information that enters the infrastructure must be encrypted. However, rather than assuming that servers are passive repositories of information (such as in CFS [6]), we want 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*. We call this policy *promiscuous caching*. Data which 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 [22] and AFS [51], in which cached data is confined to particular servers, in particular regions of the network. Experimental systems such as XFS [5] allow "cooperative caching" [17], but this is limited to systems connected by a fast local LAN.

Nomadic data lends itself to a number of "fluid" analogies: the aggregate collection of servers in the world form an "ocean" of data; this data quickly "flows" to where it is needed; individual caches could be thought of as comprising "pools" of data. This notion is illustrated in Figure 1.

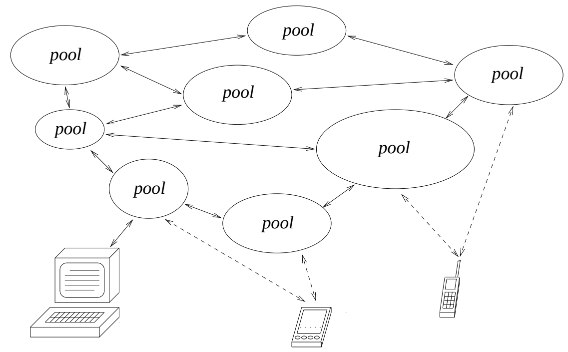


Figure 1: *The OceanStore system.* The core of the system is composed of a multitude of highly connected "pools". Clients connect to one or more pools, perhaps intermittently.

The rest of this paper is as follows: Section 2 shows sample applications of the OceanStore. Section 3 gives a system-level overview of the OceanStore system. 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 Example Applications

Before delving into the design of OceanStore, we outline three classes of applications to motivate the system and its architecture.

E-Mail: OceanStore provides a storage and distribution system that can easily replace traditional e-mail services. OceanStore users can receive their e-mail within OceanStore by reserving an OceanStore object as their "inbox". To send mail to such a user, other users simply append messages directly to the inbox object. Use of public-key cryptography permits many users to append messages to the inbox, but only permits a single person to read them. Although this style of e-mail delivery will initially require proxies to translate SMTP, POP, and IMAP requests into OceanStore requests, e-mail clients will eventually invoke OceanStore interfaces directly.

To read e-mail, a client would invoke their favorite reader program. This program would truncate messages from the beginning of the inbox and place the resulting messages into a database-like structure of folders within folders. Since this "mail repository" would be in the OceanStore, copies would be kept coherent automatically without explicit synchronization operations (such as with IMAP [16]).

OceanStore objects, unlike e-mail addresses, are not bound to specific servers. This location independence combined with replication makes the system more robust to failures and denial-of-service attacks. Barring extensive network partitioning, a user is unlikely to be out of e-mail contact for long. Note that this example utilizes two styles of consistency: weak, simultaneous updates to the e-mail inbox (as long as updates are committed "atomically") and stronger consistency for the e-mail repository.

Multimedia Applications: OceanStore provides an interesting platform for streaming multimedia applications given its efficient *append* and *truncate from beginning* semantics on objects (see Section 4.3). When objects are updated, new information is rapidly disseminated via "push-based" mechanisms. This is a good match to the fact that multimedia applications require efficient and timely delivery of new information. Note that extremely weak update semantics may be used on streams, permitting efficient pipelining of updates. Further, OceanStore's integral support of encryption permits content-for-fee applications requiring strong security guarantees.

Database Applications: As a final example, OceanStore's update mechanism can support ACID database semantics as long as the rate of conflicting transactions is reasonable. To achieve this, an application uses the OceanStore API in a form of optimistic concurrency, packaging the read and write sets into an OceanStore update packet and requesting update as long as no conflicting transactions occur. The existence of transactions provides great flexibility to OceanStore clients.

3 System Overview

The fundamental unit of information in OceanStore is the *persistent object.* Users interact with objects to store information and to communicate with other users. All naming, consistency, and access control center around objects. Since objects may contain pointers to other objects or names of other objects, they can represent directories or local name spaces (as in SDSI [48]) in addition to raw data. Hence, OceanStore provides sufficient data semantics to construct arbitrary persistent data structures.

OceanStore exploits recent gains in processor performance, storage capacity, and network bandwidth to achieve durability, stability, predictability, and security of objects while at the same time providing fast access. Redundancy and coding are used to provide durability. Caching and excess bandwidth are coupled with monitoring to yield predictable performance. Extra processing cycles are used for cryptographic protection of information. Mechanisms such as multicast are used whenever possible to push updates quickly to users. The OceanStore API seeks to provide sufficient abstraction to enable many interesting optimizations "under the covers"; the OceanStore utility exploits these opportunities.

The remainder of this section introduces some of the salient features of the OceanStore system, culminating with a description of the OceanStore API. The actual implementation of these features is explored in greater detail in Section 4.

3.1 Version Based Consistency and Deep Archival Storage

In principle, every update to an OceanStore object creates a new version[1]. Consistency based on versioning, while more expensive to implement than update-in-place consistency, provides for cleaner recovery in the face of system failures [56]. 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 [15]; 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*.

3.2 Security, Encryption, and Conflict Resolution

OceanStore supports traditional access control notions such as read and write permissions, owners, and groups. Since servers may leak or corrupt data, the underlying mechanisms used to support those policies must be significantly different from the trusted operating system code used by most systems. In general, we are concerned with two standard types of policy enforcement:

- *Restricting readers* — In order to prevent unauthorized reads, we encrypt all data in the system which is not completely public, and distribute the encryption key to those users with read permission.
- *Restricting writers* — OceanStore writes must be signed, so that well-behaved servers and clients can verify them against an access control list. Since decisions to commit data are performed by a quorum of servers, we can trust that writes are only performed if allowed.

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.

The distributed server problem is fundamentally more interesting in an untrusted infrastructure. Here it is important to enforce secrecy and integrity not only of the data in the system, but also of the meta-data and the lookup process. For instance, a user who looks up the file *ssh* must check not only that the file she receives is signed appropriately, but also that it is indeed *ssh*. Section 4.1 discusses this issue in more detail. Further, servers must be able to perform logging, commitment, and Bayou-style conflict resolution [18] *without access to cleartext or encryption keys*. Although this restricted form of computation on encrypted data might *appear* to be impossible, it is not. Section 4.3 discusses our approach.

One of the most difficult problems in OceanStore is guaranteeing access to authorized parties. Foiling denial-of-service attacks is an issue that OceanStore tackles in several ways, including redundant requests, rapid migration, and on-demand replication. Since data is nomadic and replicated, OceanStore presents no centralized target to attack.

3.3 Floating Replicas and the Update Architecture

Objects which are being read or written must be in an active state, i.e. composed of geographically distributed *floating replicas*. As shown in Figure 2, a floating replica contains a complete copy of the object's data, its meta-data, and logs of uncommitted updates. The term "floating replica" indicates the presence of all of the major components of a replica of a traditional database, while emphasizing that an OceanStore replica is not tied to a particular physical server. To activate an inactive object, OceanStore coalesces archival fragments of the latest version of this object into floating replicas at

[1] 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 [50].

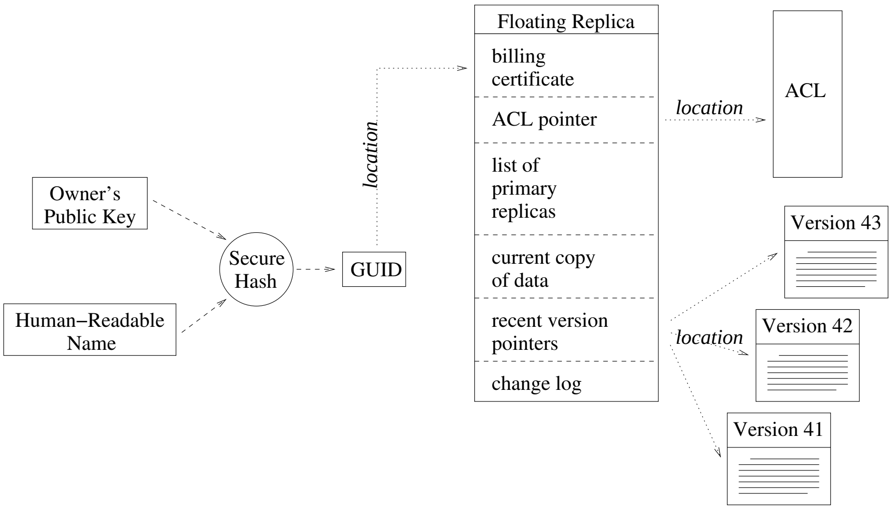


Figure 2: *An active object.* Combining a public key and human-readable name via a secure hash gives a global ID. The location mechanism take this ID and returns a pointer to a floating replica, which contains the current data and a log of client-signed updates, as well as other administrative information.

several different locations. Once an object is active, client reads and writes are directed a nearby floating replica, which may reside on the client itself. As updates are committed, the floating replicas cooperate to produce new archival copies. Should an object become inactive for a sufficiently long period of time, its floating replicas are destroyed or "reabsorbed" into the infrastructure.

Floating replicas are organized into primary and secondary tiers on a per-object basis and cooperate with one another to provide consistent update semantics (see Section 4.3). Members of the primary tier are responsible for serializing updates. They reside in the backbone of the network and communicate via a Byzantine-fault tolerant protocol, such as [12]. The secondary tier rapidly disseminates tentative commits to improve performance and survivability. Update requests are similar to updates in the Bayou system [18], in that they support conflict resolution. Through its update mechanisms, OceanStore supports a wide variety of commit semantics, from extremely weak "eventual consistency" semantics to ACID transactions.

3.4 Data Location and Introspective Optimization

Unlike other systems, persistent objects in OceanStore are free to migrate throughout the infrastructure. This capability provides important flexibility for tuning the availability, durability, and performance of object access. To support this flexibility, OceanStore employs a unique two-part data location mechanism that combines a quick, probabilistic search with a slower, guaranteed traversal of a redundant and fault-tolerant backing store (See Section 4.2). Every name, floating replica, archival fragment and access-control list is assigned a *globally unique ID* (GUID) derived from a cryptographically secure hash function. The location mechanism takes this GUID and returns a pointer to the closest copy of the corresponding object. Using a uniform lookup and caching structure allows everything, including names, ACLs, and objects, to use the same resources for caching.

Given the flexibility afforded by the naming mechanism, 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 which formalizes the automatic and dynamic optimization employed by "intelligent" systems. Introspection is used to direct many aspects of the OceanStore system, as discussed in Section 4.5.

Also, OceanStore seeks to achieve "stability through statistics", i.e. the use of many components to achieve more predictable behavior. Introspection is a major component of this strategy, dynamically adjusting system resources to adapt to denial-of-service attacks and regional outages of components.

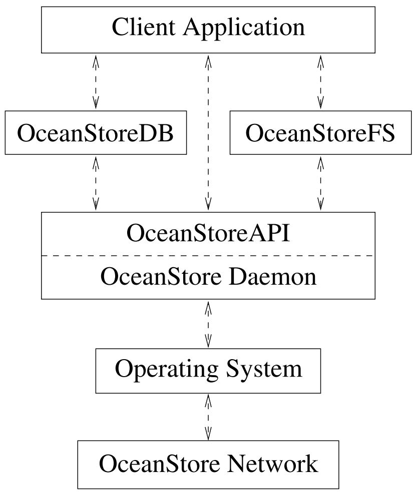


Figure 3: OceanStore client applications can choose from the native API, or special purpose APIs for database or file system semantics.

3.5 The OceanStore API

There is little expectation of uniformity of either the clients or the applications of OceanStore. Some clients may be desktop workstations with constant connectivity, while others may be personal digital assistants (PDAs) or cellular phones, which only connect intermittently. Applications might expect file system consistency semantics, require strict ACID semantics, or design custom semantics to achieve greater update performance or read availability. Finally, different types of data might have different security needs.

In response to these diverse requirements, we plan to offer several different Application Programming Interfaces (APIs) through which OceanStore can be accessed. A base API will provide access to the full set of parameters, while user-level libraries will provide traditional abstractions. Figure 3 illustrates this layering of APIs.

The following list enumerates most of the obvious parameters that an OceanStore application or library might wish to tune:

- *Consistency guarantees* — OceanStore's conflict resolution mechanisms combined with application callbacks permit a range of consistency requirements to be met.
- *Security requirements* — The level of security is appropriate for the data. The access control policy and limits to promiscuous caching can both be varied.
- *Archiving information* — Users can specify how quickly the data should be archived or forgotten.
- *Replication requirements* — Replication degree determines the level of durability and availability for the data.
- *Introspective hints* — Automatic data placement and prefetching can be assisted by users' hints.

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 lightly deployed system. Therefore, OceanStore will also provide a number of layers which implement common APIs, including Unix file system semantics and simple transactions. In addition, a gateway to the World Wide Web will permit users to access legacy documents, while still enjoying the performance advantages of promiscuous caching.

4 System Architecture

In this section, we will describe underlying technologies that support the system of Section 3. We start with *security* 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 issues involved in *consistency management* for an untrusted infrastructure. After a brief word on the architecture for *archival storage*, we finish by describing the role of *introspection* in OceanStore.

4.1 Security Issues

The untrusted infrastructure assumption provides many unique challenges to OceanStore. First and foremost, it imposes a requirement that all information which is not public be encrypted. Below, we describe the design elements of OceanStore that allow secure operation in such an infrastructure.

4.1.1 GUIDs and Naming

At the lowest level, objects in the OceanStore are identified by a *globally unique identifier* (GUID). There are two distinct classes of objects in OceanStore: read-only objects, such as archival versions, and floating replicas. The GUID of a read-only object is a secure hash over the data. Every version of every object is uniquely identified in this way. Clients requesting read-only data via its GUID can easily verify that they received the proper data by recomputing the hash. GUIDs for read-only objects are much like inodes in a traditional system, except that they are valid for all time.

For replicas, on the other hand, GUIDs must be chosen more carefully. We need a mechanism for mapping human-readable *names* to GUIDs which is decentralized and resistant to attempts by adversaries to "hijack" names which belong to other users. We solve this problem by adapting the ideas of self-certifying path names due to Mazières [41]. Replica GUIDs combine the human-readable name and the public key of the owner using a cryptographically strong hash such as SHA-1 [44]. This scheme allows servers to verify an object's owner efficiently, which facilitates access checks and resource accounting[2]. To permit verification of the mapping from the name to object data, all signatures over the data include the GUID in the input.

The above scheme does not solve the problem of generating a secure GUID mapping, but rather reduces it to a problem of secure key lookup, which we address using the locally linked name spaces from the SDSI framework [1, 48].

4.1.2 Access control

The owner of an object can securely choose the access control list (ACL) for each object. The metadata for object *foo* contains a signed certificate, which translates to "Owner says use ACL x for object *foo*". The responsible party also signs the certificate, along with its expiration date. The specified ACL may be another object, or may be a value indicating a common default.

An ACL entry extending privileges must describe the privilege granted and the signing key, but not 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 [2, 3, 7, 35].

Restricting read access is entirely a matter of restricting access to an encryption key. To revoke read permission, the owner must request that replicas be deleted or resigned with the new key. A recently-revoked reader will be able to read old data from cached copies or from misbehaving servers which 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.

[2] 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 they share with various groups.

4.1.3 Admission control

Clients and servers may not regard all other participants in the OceanStore as equals. A business may want OceanStore services internally, but not allow external information on their servers. Or it may want to ensure that critical data never leaves physically secure servers owned by the company. In the "ocean of data" metaphor, we need some way to create guarded "bays". Therefore, we allow servers to enforce a local policy for admission and export of data.

OceanStore attempts to tackle denial-of-service attacks on several fronts. Underlying everything is a strict accounting of resource usage. In principle, OceanStore has sufficient accounting to track and reject excessive requests from a single source. In addition, OceanStore employs intelligent fault recognition and automatic adaptation to heavy loads; see Section 4.5.

4.2 Data Location

The role of data location is to provide a mapping from GUIDs to floating replicas, access control lists, or versions of data objects. The mechanism is a two-tiered approach featuring a fast, probabilistic algorithm backed up by a slower, reliable hierarchical method. The local algorithm returns objects rapidly if they are locally available. A large-scale hierarchical data structure in the style of Plaxton et. al. [46] locates objects that cannot be found locally. This data location mechanism is used uniformly throughout the system along with secure pointers to validate that the returned object is correct.

4.2.1 The Local Algorithm

The local 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 filters [8]—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

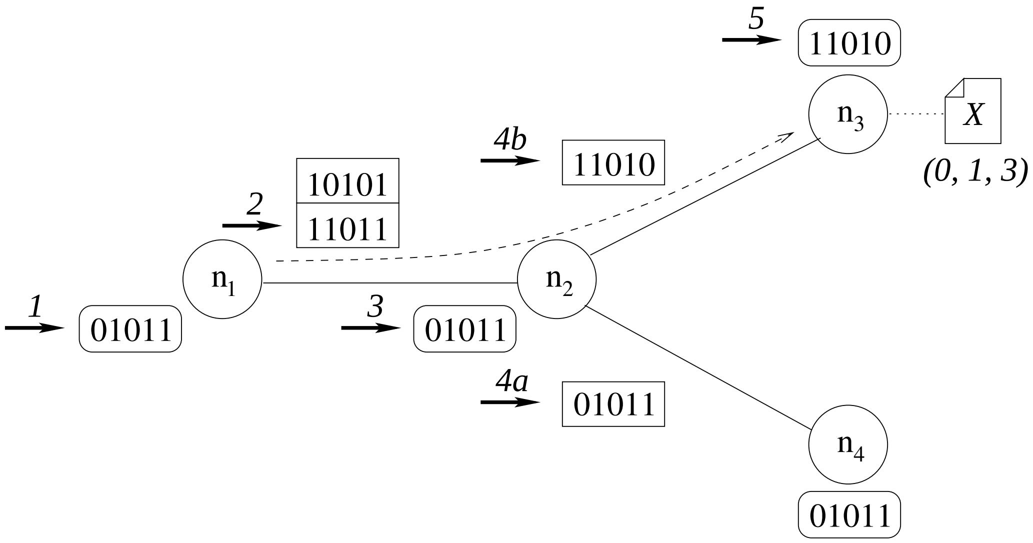


Figure 4: *The local query process.* The replica at n_1 is looking for object X, whose GUID hashes to bits 0, 1, and 3. (1) The local Bloom filter for n_1 (rounded box) shows that it does not have the object, but (2) its neighbor filter (unrounded box) for n_2 indicates that n_2 might be an intermediate 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.

a record of the object contained locally on the current node. The ith Bloom filter is the merger 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 4. 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, "trust factors" can be applied locally to increase the distance to nodes which have abused the protocol in the past, automatically routing around certain classes of attacks.

4.2.2 The Global Algorithm

The global algorithm for the OceanStore is a variation on Plaxton et. al.'s randomized hierarchical distributed data structure, which embeds a tree in the network and uses a prefix-based labeling scheme and per-node neighbor tables to maintain a good path from every node to every object. Requests are satisfied optimally with high probability and document motion updates require $O(\log n)$ messages in the number of servers.

The algorithm is sensitive to a number of different failures, so we introduce a modified version for use in OceanStore. A small, constant number of labeling schemes can be used to decrease the dependency on a single root. Since true updates are expensive, the backing store can profit by storing forwarding pointers to regions in the OceanStore where a local search is guaranteed to find the object. This allows small-scale object migration without requiring expensive tree updates.

4.3 Update Model

The OceanStore model of updates involves *conflict resolution*, as introduced in the Bayou system [18]. Conflict resolution supports a wide range of consistency semantics, but requires the ability to perform server-side computations on data. In an untrusted infrastructure, servers have access only to ciphertext, and no one server is trusted to perform commits. Both of these issues complicate the update architecture. In the following paragraphs, we describe the issues and our progress towards solving them.

4.3.1 The Bayou Mechanism

Application specific conflict resolution as presented in Bayou provides a mechanism for the automatic handling of update conflicts in a weakly-consistent storage system. Each update consists of three parts: a dependency check, an update set, and a merge procedure. To apply an update, a server first runs the dependency check against its version of the given object. If the check succeeds, the update set is applied to the object; otherwise, the merge procedure is executed to resolve the conflict.

Updates are propagated between servers in the system via a process called *anti-entropy*, which is similar to log shipping [43, 25] in a replicated database. Until an update reaches a specific server, called the *primary server*, it is considered *tentative*. The primary server serializes the update with respect to other updates on the same data object; at that point the update is said to have *committed*.

Bayou supported a number of applications, including a group calendar, a shared bibliographic database, and a mail application [19]. However, the model can be applied more generally. For instance, Coda [31] provided specific merge procedures for conflicting updates of directories; this type of conflict resolution is easily supported under the Bayou model. Slight extensions to the Bayou model can support Lotus Notes-style conflict resolution, where unresolvable conflicts result in a branch in the object's version stream [30]. Finally, the Bayou model can be used to provide ACID semantics: the predicate becomes the read set of a transaction, the update becomes the write set, and the merge predicate always fails[3].

4.3.2 Extending the Bayou Mechanism to Work over Ciphertext

In OceanStore, servers are not trusted with unencrypted information. This complicates all three server-side Bayou operations: *computing predicates*, *applying modifications*, and *performing merge*. Since arbitrary computations and manipulations on ciphertext are still intractable, this leads us to abandon the notion of a general merge procedure, using a list of alternate predicate-update pairs instead. Fortunately, a number of powerful but specialized operations *can* be applied to encrypted data. For instance, the following predicates are currently possible:

[3] This is similar to optimistic concurrency control as discussed in [34].

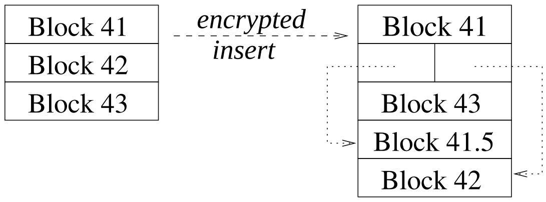


Figure 5: *Block insertion on ciphertext.* The client wishes to insert block 41.5, so she appends it and block 42 to the object, then replaces the old block 42 with a block pointing to the two appended blocks. The server learns nothing about the contents of any of the blocks.

- *compare-version* (*version-id*) — Check that the version of this object is greater than *version-id*.
- *compare-size* (*size*) — Compare the size of this object to *size*.
- *compare-block* (*block-no*, *expected-value*) — Compare a block of an object against a given expected value.
- *search* (*search-string*) — Does the given string occur within the object? If so, how many times does it occur?

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 encrypts the given 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 [54]; this operation reveals only that a search was performed along with the encrypted result. The cleartext of the search string is not revealed, nor can the server initiate new searches on its own.

In addition to these predicates, the following operations can be applied to ciphertext:

- *replace-block* (*block-no*, *new-value*) — Modify a block to have the given new value.
- *insert-block* (*block-no*, *value*) — Insert a new block into the object.
- *delete-block* (*block-no*) — Remove the given block from the object.
- *append* (*value*) — Append the given block to the object. (This operation and the next are useful for streaming media data objects.)

- *truncate-from-start* (*num-blocks*) — Remove the first *num-blocks* from an object.

Again assuming a position-dependent block cipher, the *replace-block* and *append* operations are simple for the same reasons as *compare-block*. The *truncate-from-start* operation is trivial, leaving only the *insert-block* and *delete-block* operations.

These last two operations can be performed by grouping all blocks of the object into two sets, index blocks and data blocks, where index blocks contain pointers to other blocks elsewhere in the object, similar to *inodes* in the Unix FS [42]. Insertion involves replacing 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. Deletion involves replacing the block in question with an empty pointer block. This scheme is illustrated in Figure 5. Note that it leaks a small amount of information and thus might be susceptible to traffic-analysis; 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.

4.3.3 Serializing Updates in an Untrusted Infrastructure

Bayou relies on a single primary server to pick its final order of updates. Unfortunately, trusting any one replica to perform this task is incompatible with the untrusted infrastructure on which OceanStore is based. Thus, we replace this primary server with a primary *tier* of floating replicas. These replicas cooperate with one another via a Byzantine fault-tolerant algorithm to choose the final commit order for updates. A secondary tier of replicas communicates amongst 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 tolerant 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[4]. Since existing protocols for serialization by a group of replicas do not include provisions for later off-line verification by a party who did not participate in the protocol, we are exploring techniques like verifiable secret sharing [11] to allow off-line auditing of the serialization process. To this end, our group is currently investigating extending the protocol in [12] to allow for quick off-line validation.

Some applications may gain performance or availability by requiring a lesser degree of consistency than ACID semantics. These applications are well supported by the secondary tier of replicas in OceanStore. Secondary replicas can be more numerous than primary replicas and exist at the leaves of the network. They make use of multicast or other transport mechanisms to quickly *push* tentative commits amongst themselves and to decide a tentative serialization order. This property aids streaming applications, since data is rapidly and efficiently distributed to replicas. Since the serialization decisions of the secondary tier are tentative, they may be safely decided by untrusted servers; applications requiring stronger consistency guarantees must simply wait for their updates to reach the primary replicas. We intend to provide session guarantees for individual clients, much in the flavor of Bayou [57].

4.4 Deep Archival Storage

The archival mechanism of OceanStore employs *erasure codes*, including interleaved Read-Solomon codes [45] and Tornado codes [37]. Erasure coding is a process which treats input data as a series of fragments (say n) and transforms these fragments into a greater number of fragments (say $2n$ or $4n$). The essential property of the resulting code is that *any* n of the coded fragments are sufficient to construct the original data[5]. 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 arrangement *deep archival storage*.

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 [50].

New archival copies are generated at regular intervals and after a user-selected number of updates. Although in principle every version of every object is

[4] We also assume that the client's responsible party participates in the primary tier and so assists in guaranteeing commitment.

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

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 send this subset into the infrastructure.

To maximize the survivability of archival copies, we identify and rank administrative domains by their reliability and trustworthiness. Of key importance is to 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) will be determined on a per-object basis. As mentioned in Section 4.5.3, OceanStore contains processes which slowly sweep through all existing archival data, repairing or increasing the level of replication to further increase durability.

To reconstruct archival copies, OceanStore consults the global location index (Section 4.2), which stores the location of every fragment. The search for fragments is keyed off the GUID of archival versions. This search has nice locality properties since closer fragments tend to be discovered first. When activating an inactive object, we initiate search in several different locales simultaneously, one for each floating replica. 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.

4.5 Introspection

The effort required to optimize a system increases with its size and complexity. As envisioned, OceanStore would consist of millions of individual servers, each of which differ in connectivity, disk capacity, and computational power. New servers or disk devices will come online sporadically. The loads on individual servers will vary from moment to moment. Hence, manually tuning a system as large and varied as OceanStore is completely implausible. Worse, in the utility model, manual tuning would involve cooperation between competing administrative domains.

To address this problem, OceanStore employs *introspection*, an architectural paradigm that mimics adaptation in biological systems. As shown in Figure 6, introspection augments system functionality (*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. Introspection generates non-linear feedback that can be directed to enhance or dampen system variations.

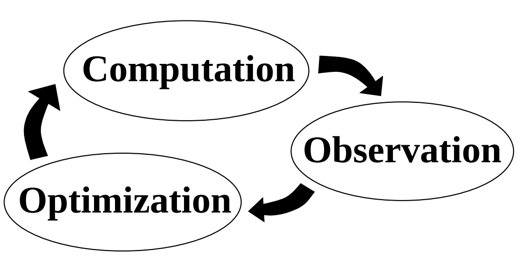


Figure 6: The Cycle of Introspection

OceanStore uses introspection to select the placement and number of floating replicas, to prefetch information, and to monitor and repair the level of redundancy. Introspection also helps to improve the quality of service. Although we have insufficient space to describe these in detail, we will give a flavor for our techniques in the following paragraphs.

4.5.1 Adaptive Replica Management

Both the number and location of floating replicas greatly affect availability and performance. A high degree of replication increases availability and decrease access latency. For read-only objects, this is clearly desirable. For frequently written objects, however, each replica increases the effort and overhead required to maintain consistency. An optimal placement and degree of replication, even if it could be computed, would vary from moment to moment as clients moved or changed their behavior.

In OceanStore, each floating replica tracks client requests and measures the server load consumed when servicing these requests. This information is distilled and used to decide whether the floating replica should migrate closer to the source of client requests or toward the centroid of a group of updating clients. The information is also used to decide when a floating replica should spawn a copy of itself or be destroyed. At the same time, servers extract knowledge of the network topology from routing information collected during data location. This information is used to select which neighboring server will receive new replicas.

Unfortunately, compromised servers may wreck havoc by initiating excessive migration or by improperly destroying floating replicas. OceanStore takes a conservative approach in preventing adversaries from spoofing introspection. First, gratuitous migration is masked through hysteresis. Further, we ignore requests not properly authenticated by clients and are careful to gather and contrast information from a number of different sources before we act on it. Finally, serious decisions such as the destruction of a replica involve agreement by a quorum of its peers.

4.5.2 Prefetch and Pro-Active Migration

Access to distant data in OceanStore could incur high latency. We are investigating two categories of data prefetch to ameliorate this latency: on-demand prefetch of *clusters* of related objects and *pro-active migration* of objects (or clusters) before they are needed. By migrating complete clusters of related objects, OceanStore seeks to improve average access time. A simple example is the set of pages linked from a given web page; some of these will be accessed after the base page with high frequency. Clustering these pages exploits this semantic information in order to increase prefetch coverage.

Each client includes introspective monitoring that examines the client's access stream to determines the *semantic distance* [33] between objects as accessed by a given client; semantic distance is a metric that attempts to capture the degree of "relatedness" between different objects. This information is collected over time for a better characterization of access behavior[6]. The result of this monitoring is a weighted, directed graph of distances between objects. This graph is partitioned by a clustering algorithm to discover clusters of related objects.

Other introspective monitors track access patterns to clusters as a whole. These patterns are analyzed via time-series techniques such as Hidden Markov Models or Kalmann filters to discover cluster orbits (repeated patterns over time). Examples include users who spend their weekdays at work, evenings at home, and weekends at a cabin. Their data should be waiting for them when they arrive at each new venue. We hope to extract such patterns and migrate data to places where it will be needed in the future.

4.5.3 Stability and Durability Enhancement

If not properly checked, nearly any optimization in the OceanStore can cause instability. In one scenario, data could oscillate wildly from place to place, resulting in a decreased quality of service. Under high load, the possibility of a "data brownout" arises. We are investigating ways in which monitoring can make global adjustments to stabilize mean response time and variance. This is a form of negative feedback for stabilization.

In addition, the durability of data stored in the OceanStore is imperative. The design of the system ensures that data is protected from a reasonable numbers of failures and localized attacks. To increase durability further, OceanStore includes introspective mechanisms which perform "sweep and repair" functionality. Two key subsystems are targeted by this functionality: the data location mechanism, and the archival storage mechanism. For the global data location structure, we continually rebuild the search tree and pointers for data. For archival data, OceanStore servers slowly sweep through all fragments, increasing the level of replication when necessary.

[6] There is an interesting tradeoff here between privacy and optimization benefits; we are still exploring these issues.

5 Status

We are currently implementing an OceanStore prototype, designed for "read-mostly" workloads, which we will deploy for testing and evaluation. The system is written in Java, using Jaguar [59] and a state machine-based request model for fast I/O. Initially, OceanStore will communicate with legacy applications both through a UNIX file system interface and a read-only proxy for the World Wide Web.

We have explored the requirements which 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 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 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, which use both Reed-Solomon and Tornado codes for redundancy encoding. Although only one half of the fragments were required to reconstruct the object, 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 for noise. That mechanism will be combined with an optimization module appropriate for the wide-area network.

6 Related Work

Distributed systems such as Taos [3, 35] assume untrusted networks and applications, but rely on some trusted computing base. Cryptographic file systems such as Blaze's CFS [6] provide end-to-end secrecy, but include no provisions for sharing data, nor for protecting integrity independently from secrecy. The Secure File System [29] supports sharing with access control lists, but fails to provide independent support for integrity, and trusts a single server to distribute encryption keys.

SDSI [1, 48] and SPKI [20] address the problem of securely distributing keys and certificates in a decentralized manner. PolicyMaker [7] deals with the description of trust relations. Mazières proposes self-certifying paths to separate key management from system security [40, 41].

Bloom filters [8] are commonly used as compact representations of large sets. The R* distributed database [38] calculates them on demand to implement efficient semijoins. The Summary Cache [21] pushes Bloom filters between cooperating web caches, although their method does not scale well.

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 [36]. In particular, Bayou [18] and Coda [31] use replication to improve availability at the expense of consistency and introduce specialized conflict resolution procedures. Neither system addresses the range of security concerns that OceanStore does, although Bayou examines some problems that occur when replicas are corrupted [55].

Gray et. al. argue against promiscuous replication in [26]. 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 class of floating replicas are similar to transactional caches; in the taxonomy of [23] our algorithm is detection-based and performs its validity checks at commit time. In contrast to such systems, our merge predicates decrease the number of transactions aborted due to out-of-date caches.

Many previous projects have explored feedback–driven adaptation in extensible operating systems [52], databases [13, 14], file systems [39], global operating systems [9], and storage devices [10, 60]. 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 [33] and collects clusters of related files for automated hoarding. Others have used file system observation to drive automatic prefetching [27, 32].

Introspective replica management for web content was examined in AT&T's Radar project [47], which considers read-only data in a trusted infrastructure. The Mariposa project [53] addresses inter-domain replication with an economic model. Others optimize communication cost when selecting a new location for replica placement [4] within a single administrative domain.

The Intermemory project [24] uses Cauchy Reed-Solomon Codes in a distributed storage system. Such codes tolerate fewer simultaneous failures than those used in OceanStore. Additionally, we anticipate that our combination of active and archival object forms will allow greater update performance than such systems, while retaining their 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.

Two properties distinguish OceanStore from other systems: the *untrusted infrastructure* and support for truly *nomadic data*. 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 in which information access can be optimized by bringing it "close" to where it is needed. These optimizations are assisted by *introspection*, the continuous online collection and analysis of access patterns. Nomadic data also enables rapid response to regional outages and denial-of-service attacks.

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.

References

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

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM TOPLAS*, 15(4):706–734, Sept. 1993.

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

[4] 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.

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

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

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

[8] 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.

[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] E. Borowsky, R. Golding, A. Merchant, E. Shriver, M. Spasojevic, and J. Wilkes. Eliminating storage headaches through self-management. In *Proc. of USENIX Symp. on OSDI*, Oct. 1996.

[11] M. Burmester. Homomorphisms of sharing schemes: a tool for verifiable signature sharing. In *Advances in Cryptology (EUROCRYPT)*, pages 96–106, 1996.

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

[13] S. Chaudhuri and V. Narasayya. An efficient, cost-driven index selection tool for Microsoft SQL server. In *Proc. of Intl. Conf. on VLDB*, pages 146–155, Aug. 1997.

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

[15] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. Prototype implementation of archival intermemory. In *Proc. of IEEE ICDE*, pages 485–495, Feb. 1996.

[16] M. Crispin. Internet message access protocol — version 4rev1. RFC 2060, Dec. 1996.

[17] 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.

[18] 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*, pages 2–7, Dec. 1994.

[19] 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.

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

[21] 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.

[22] I. E. T. Force. NFS network file system protocol specification. In *RFC 1094*, Mar. 1989.

[23] 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.

[24] A. Goldberg and P. Yianilos. Prototype implementation of archival intermemory. In *Proc. of IEEE ADL*, pages 147–156, Apr. 1998.

[25] A. Gorelik, Y. Wang, and M. Deppe. Sybase replication server. In *Proc. of ACM SIGMOD Conf.*, May 1994.

[26] 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.

[27] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. Technical Report CS247-94, U. of Kentucky, June 1994.

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

[29] 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.

[30] 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.

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

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

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

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

[35] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM TOCS*, 10(4):265–310, Nov. 1992.

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

[37] 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.

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

[39] 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.

[40] D. Mazières and F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *Proc. of ACM SIGOPS*, Sept. 1998.

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

[42] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. *ACM TOCS*, 2(3):181–197, Aug. 1984.

[43] A. Moissis. SYBASE replication server: A practical architecture for distributing and sharing corporate information. Technical report, SYBASE Inc., Mar. 1994.

[44] N. I. of Standard and Technology. Digital signature standard. Technical Report 186, US Department of Commerce, May 1994.

[45] J. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. http://cs.utk.edu/˜ plank, Feb. 1999.

[46] 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.

[47] 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.

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

[49] J. Rothnie. Overview of the KSR1 computer system. Technical Report TR 9202001, Kendall Square Research, Mar. 1992.

[50] 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.

[51] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

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

[53] 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.

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

[55] 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.

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

[57] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. of Intl. Conf. on PDIS*, pages 140–149, Sept. 1994.

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

[59] M. Welsh and D. Culler. Jaguar: Enabling efficient communication and I/O from Java. *Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications*, Dec. 1999.

[60] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM TOCS*, pages 108–136, Feb. 1996.