Appears in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (FAST '03).

Pond: the OceanStore Prototype*

Sean Rhea, Patrick Eaton, Dennis Geels,
Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz
University of California, Berkeley
{*srhea,eaton,geels,hweather,ravenben,kubitron*}@*cs.berkeley.edu*

Abstract

OceanStore is an Internet-scale, persistent data store designed for incremental scalability, secure sharing, and long-term durability. Pond is the OceanStore prototype; it contains many of the features of a complete system including location-independent routing, Byzantine update commitment, push-based update of cached copies through an overlay multicast network, and continuous archiving to erasure-coded form. In the wide area, Pond outperforms NFS by up to a factor of 4.6 on read-intensive phases of the Andrew benchmark, but underperforms NFS by as much as a factor of 7.3 on write-intensive phases. Microbenchmarks show that write performance is limited by the speed of erasure coding and threshold signature generation, two important areas of future research. Further microbenchmarks show that Pond manages replica consistency in a bandwidth-efficient manner and quantify the latency cost imposed by this bandwidth savings.

1 Introduction

One of the dominant costs of storage today is management: maintaining the health and performance characteristics of data over the long term. Two recent trends can help ameliorate this problem. First, the rise of the Internet over the last decade has spawned the advent of universal connectivity; the average computer user today is increasingly likely to be connected to the Internet via a high-bandwidth link. Second, disk storage capacity per unit cost has skyrocketed; assuming growth continues according to Moore's law, a *terabyte* of EIDE storage will cost $100 US in under three years. These trends present a unique opportunity for file system designers: for the first time, one can imagine providing truly durable, self-maintaining storage to every computer user.

OceanStore [14, 26] is an Internet-scale, cooperative file system designed to harness these trends to provide high durability and universal availability to its users through a two-tiered storage system. The upper tier in this hierarchy consists of powerful, well-connected hosts which serialize changes and archive results. The lower tier, in contrast, consists of less powerful hosts—including users' workstations—which mainly provide storage resources to the system. Dividing the system into two tiers in this manner allows for powerful, well-connected hosts to provide services that demand many resources, while at the same time harnessing the vast storage resources available on less powerful or less well-connected hosts.

The unit of storage in OceanStore is the *data object*, onto which applications map more familiar user interfaces. For example, Pond includes both an electronic mail application and a UNIX file system. To allow for the greatest number of potential OceanStore applications, we place the following requirements on the object interface. First, information must be universally accessible; the ability to read a particular object should not be limited by the user's physical location. Second, the system should balance the tension between privacy and information sharing; while some applications demand the ability to aggressively read- and write-share data between users, others require their data to be kept in the strictest confidence. Third, an easily understandable and usable consistency model is crucial to information sharing. Fourth, privacy complements integrity; the system should guarantee that the data read is that which was written.

With this interface in mind, we designed OceanStore under the guidance of two assumptions. First, the infrastructure is untrusted except in aggregate. We expect hosts and routers to fail arbitrarily. This failure may be passive, such as a host snooping messages in attempt to violate users' privacy, or it may be active, such as a host injecting messages to disrupt some protocol. In aggregate, however, we expect hosts to be trustworthy; specifically, we often assume that no more than some fraction of a given set of hosts are faulty or malicious.

A second assumption is that the infrastructure is constantly changing. The performance of existing communication paths varies, and resources continually en-

*Research supported by NSF career award #ANI-9985250, NFS ITR award #CCR-0085899, and California MICRO award #00-049. Dennis Geels is supported by the Fannie and John Hertz Foundation.

Name	Meaning	Description
BGUID	block GUID	secure hash of a block of data
VGUID	version GUID	BGUID of the root block of a version
AGUID	active GUID	names a complete stream of versions

Table 1: *Summary of Globally Unique Identifiers (GUIDs).*

ter and exit the network, often without warning. Such constant flux has historically proven difficult for administrators to handle. At a minimum, the system must be self-organizing and self-repairing; ideally, it will be self-tuning as well. Achieving such a level of adaptability requires both the redundancy to tolerate faults and dynamic algorithms to efficiently utilize this redundancy.

The challenge of OceanStore, then, is to design a system which provides an expressive storage interface to users while guaranteeing high durability atop an untrusted and constantly changing base. In this paper, we present Pond, the OceanStore prototype. This prototype contains most of the features essential to a full system; it is built on a self-organizing location and routing infrastructure, it automatically allocates new replicas of data objects based on usage patterns, it utilizes fault-tolerant algorithms for critical services, and it durably stores data in erasure-coded form. Most importantly, Pond contains a sufficiently complete implementation of the OceanStore design to give a reasonable estimate of the performance of a full system.

The remainder of this paper is organized as follows. We present the OceanStore interface in Section 2, followed by a description of the system architecture in Section 3. We discuss implementation details particular to the current prototype in Section 4, and in Sections 5 and 6 we discuss our experimental framework and performance results. We discuss related work in Section 7, and we conclude in Section 8.

2 Data Model

This section describes the OceanStore *data model*—the view of the system that is presented to client applications. This model is designed to be quite general, allowing for a diverse set of possible applications—including file systems, electronic mail, and databases with full ACID (atomicity, consistency, isolation, and durability) semantics. We first describe the storage layout.

2.1 Storage Organization

An OceanStore *data object* is an analog to a file in a traditional file system. These data objects are ordered sequences of read-only versions, and—in principle—every version of every object is kept forever. Versioning simplifies many issues with OceanStore's caching and replication model. As an additional benefit, it allows for *time travel*, as popularized by Postgres [34] and the Elephant File System [30]; users can view past versions of a file or directory in order to recover accidentally deleted data.

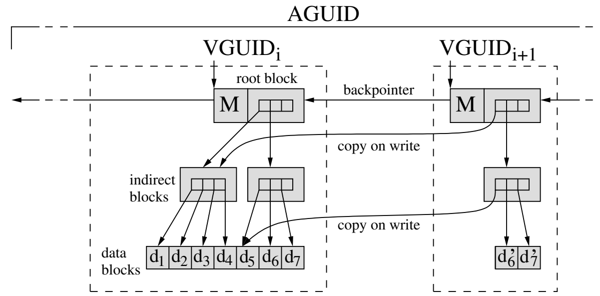

Figure 1: A data object is a sequence of read-only versions, collectively named by an *active* GUID, or AGUID. Each version is a B-tree of read-only blocks; child pointers are secure hashes of the blocks to which they point and are called *block* GUIDs. User data is stored in the leaf blocks. The block GUID of the top block is called the *version* GUID, or VGUID. Here, in version $i+1$, only data blocks 6 and 7 were changed from version i, so only those two new blocks (and their new parents) are added to the system; all other blocks are simply referenced by the same BGUIDs as in the previous version.

Figure 1 illustrates the storage layout of a data object. Each version of an object contains metadata, the actual user-specified data, and one or more references to previous versions. The entire stream of versions of a given data object is named by an identifier we call its *active globally-unique identifier*, or AGUID for short, which is a cryptographically-secure hash of the concatenation of an application-specified name and the owner's public key. Including this key securely prevents namespace collisions between users and simplifies access control.

To provide secure and efficient support for versioning, each version of a data object is stored in a data structure similar to a B-tree, in which a block references each child by a cryptographically-secure hash of the child block's contents. This hash is called the *block GUID*, or BGUID, and we define the version GUID, or VGUID, to be the BGUID of the top block. When two versions of a data object share the same contents, they reference the same BGUIDs; a small difference between versions requires only a small amount of additional storage. Because they are named by secure hashes, child blocks are read-only. It can be shown that this hierarchical hashing technique produces a VGUID which is a cryptographically-secure hash of the entire contents of a version [20]. Table 1 enumerates the types of GUIDs in the system.

2.2 Application-specific Consistency

In this section, we describe the consistency mechanisms provided to readers and writers of data objects. We define an *update* to be the operation of adding a new version to the head of the version stream of one or more data objects. In OceanStore, updates are applied atomically and are represented as an array of potential actions each guarded by a predicate. This choice was inspired by the Bayou system [8]. Example actions include replacing a set of bytes in the object, appending new data to the end of the object, and truncating the object. Example predicates include checking the latest version number of the object and comparing a region of bytes within the object to an expected value.

Encoding updates in this way allows OceanStore to support a wide variety of application-defined consistency semantics. For example, a database application could implement optimistic concurrency control with ACID semantics by letting the predicate of each update check for changes in the read set, and if none are found, applying the write set in the update's action. In contrast, the operation of adding a message to a mailbox stored as a data object could be implemented as an append operation with a vacuous predicate. One important design decision in OceanStore was not to support explicit locks or leases on data, and to instead rely on our update model to provide consistency; if necessary, the atomicity of our updates allows locks to be built at the application layer.

Along with predicates over updates, OceanStore allows client applications to specify predicates over reads. For example, a client may require that the data of a read be no older than 30 seconds, it may require the most-recently written data, or it may require the data from a specific version in the past.

3 System Architecture

We now discuss the architecture of the OceanStore system that implements the application-level interface of the previous section. The unit of synchronization in OceanStore is the data object. Consequently, although changes to a particular object must be coordinated through shared resources, *changes to different objects are independent*. OceanStore exploits this inter-object parallelism in order to achieve scalability; adding additional physical components allows the system to support more data objects.

3.1 Virtualization through Tapestry

OceanStore is constructed from interacting resources (such as permanent blocks of storage or processes managing the consistency of data). These resources are *virtual* in that they are not permanently tied to a particular piece of hardware and can move at any time. A virtual resource is named by a *globally unique identifier* (GUID) and contains the state required to provide some service. For caches or blocks of storage, this state is the data itself. For more complicated services, this state involves things like history, pending queues, or commit logs.

Virtualization is enabled by a decentralized object location and routing system (DOLR) called Tapestry [12]. Tapestry is a scalable overlay network, built on TCP/IP, that frees the OceanStore implementation from worrying about the location of resources. Each message sent through Tapestry is addressed with a GUID rather than an IP address; Tapestry routes the message to a physical host containing a resource with that GUID. Further, Tapestry is locality aware: if there are several resources with the same GUID, it locates (with high probability) one that is among the closest to the message source.

Both hosts and resources are named by GUIDs. A physical host *joins* Tapestry by supplying a GUID to identify itself, after which other hosts can *route* messages to it. Hosts *publish* the GUIDs of their resources in Tapestry. Other hosts can then route messages to these resources. Unlike other overlay networks, Tapestry does not restrict the placement of resources in the system. Of course, a node may *unpublish* a resource or *leave* the network at any time.

3.2 Replication and Consistency

A data object is a sequence of read-only versions, consisting of read-only blocks, securely named by BGUIDs. Consequently, the replication of these blocks introduces no consistency issues; a block may be replicated as widely as is convenient, and simply knowing the BGUID of a block allows a host to verify its integrity. For this reason, OceanStore hosts publish the BGUIDs of the blocks they store in Tapestry. Remote hosts can then read these blocks by sending messages addressed with the desired BGUIDs through Tapestry.

In contrast, the mapping from the name of a data object (its AGUID) to the latest version of that object (named by a VGUID), may change over time as the file changes. To limit consistency traffic, OceanStore implements primary-copy replication [10]. Each object has a single *primary replica*, which serializes and applies all updates to the object and creates a digital certificate mapping an AGUID to the VGUID of the most recent version. The certificate, called a *heartbeat*, is a tuple containing an AGUID, a VGUID, a timestamp, and a version sequence number. In addition to maintaining the AGUID to latest VGUID mapping, the primary replica also enforces access control restrictions and serializes concurrent updates from multiple users.

To securely verify that it receives the latest heartbeat for a given object, a client may include a nonce in its signed request; in this case the resulting response from the primary replica will also contain the client's name and nonce and be signed with the primary's key. This procedure is rarely necessary, however, since common applications can tolerate somewhat looser consistency semantics. Our NFS client, for example, only requests new heartbeats which are less than 30 seconds old.

We implement the primary replica as a small set of cooperating servers to avoid giving a single machine complete control over a user's data. These servers, collectively called the *inner ring*, use a Byzantine-fault-tolerant protocol to agree on all updates to the data object and digitally sign the result. This protocol allows the ring to operate correctly even if some members fail or behave maliciously. The inner ring implementation is discussed in detail in Section 3.6. The primary replica is a virtual resource, and can be mapped on a variety of different physical servers during the lifetime of an object. Further, the fact that objects are independent of one another provides maximal flexibility to distribute primary replicas among physical inner ring servers to balance load.

In addition to the primary replica, there are two other types of resources used to store information about an object: archival fragments and secondary replicas. These are mapped on different OceanStore servers from those handling the inner ring. We discuss each in turn.

3.3 Archival Storage

While simple replication provides for some fault tolerance, it is quite inefficient with respect to the total storage consumed. For example, by creating two replicas of a data block, we achieve tolerance of one failure for an addition 100% storage cost. In contrast, *erasure codes* achieve much higher fault tolerance for the same additional storage cost.

An erasure code [2] is a mathematical technique by which a block is divided into m identically-sized *fragments*, which are then encoded into n fragments, where $n > m$. The quantity $r = \frac{m}{n} < 1$ is called the *rate* of encoding. A rate r code increases the storage cost by a factor of $\frac{1}{r}$. The key property of erasure codes is that the original object can be reconstructed from *any* m fragments. For example, encoding a block using a rate $\frac{1}{2}$ code and $m = 16$ produces 32 fragments, any arbitrary 16 of which are sufficient to reconstruct the original block. Intuitively, one can thus see that erasure encoding produces far higher fault tolerance for the storage used than replication. A detailed analysis confirming this intuition can be found in our earlier work [36]. In the prototype, we use a Cauchy Reed-Solomon code [2] with $m = 16$ and $n = 32$.

Erasure codes are utilized in OceanStore as follows. After an update is applied by the primary replica, all newly created blocks are erasure-coded and the resulting fragments are distributed to OceanStore servers for storage. Any machine in the system may store archival fragments, and the primary replica uses Tapestry to distribute the fragments uniformly throughout the system based on a deterministic function of their fragment number and the BGUID of the block they encode.[1] To reconstruct a block at some future time, a host simply uses Tapestry to discover a sufficient number of fragments and then performs the decoding process.

3.4 Caching of Data Objects

Erasure coding, as we have seen, provides very high durability for the storage used. However, reconstructing a block from erasure codes is an expensive process; at least m fragments must be recovered, and assuming that the fragments of a block were stored on distinct machines for failure independence, this recovery requires the use of m distinct network cards and disk arms.

To avoid the costs of erasure codes on frequently-read objects, OceanStore also employs whole-block caching. To read a block, a host first queries Tapestry for the block itself; if it is not available the host then retrieves the fragments for the block using Tapestry and performs the decoding process. In either case, the host next publishes its possession of the block in Tapestry; a subsequent read by a second host will find the cached block through the first. Thus the cost of retrieval from the archive is amortized over all of the readers. Importantly, reconstructed blocks are only soft state; since they can be reconstructed from the archive at any time (for some cost), they can be discarded whenever convenient. This soft-state nature of reconstructed blocks allows for caching decisions to be made in a locally greedy manner (Pond uses LRU).

For reading a particular version of a document, the technique described in the previous paragraph is sufficient to ensure a correct result. However, often an application needs to read the *latest* version of a document. To do so, it utilizes Tapestry to retrieve a heartbeat for the object from its primary replica. This heartbeat is a signed and dated certificate that securely maps the object's AGUID to the VGUID of its latest version.

OceanStore supports efficient, push-based update of the secondary replicas of an object by organizing them into an application-level multicast tree. This tree, rooted at the primary replica for the object, is called the *dissemination tree* for that object. Every time the primary

[1] While using Tapestry in this manner yields some degree of failure independence between fragments encoding the same block, it is preferable to achieve this independence more explicitly. We have a proposal for doing so [37], but it is not yet implemented.

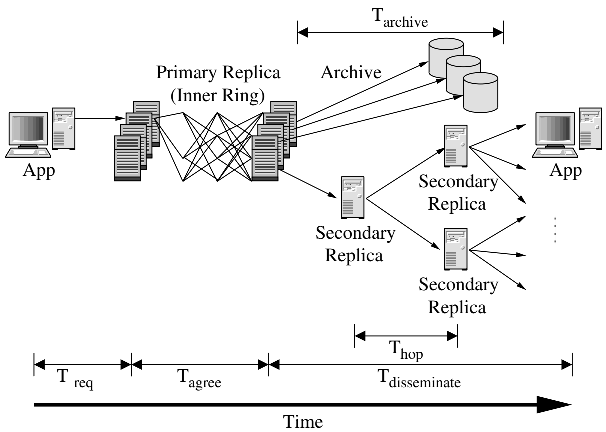

Figure 2: *The path of an OceanStore update.* An update proceeds from the client to the primary replica for its target data object. There, it is serialized with other updates and applied to that target. A heartbeat is generated, certifying the new latest version, and multicast along with the update down the dissemination tree to other replicas. Simultaneously, the new version is erasure-coded and sent to archival storage servers.

replica applies an update to create a new version, it sends the corresponding update and heartbeat down the dissemination tree. Updates are thus multicast directly to secondary replicas. The dissemination tree is built in a self-organizing fashion; each new secondary replica utilizes Tapestry to locate a nearby, pre-existing replica to serve as a parent in this tree. A more sophisticated version of this algorithm is examined elsewhere [5].

3.5 The Full Update Path

In this section, we review the full path of an update. We will postpone the description of the primary replica until the next section.

Figure 2 shows the path of an update in OceanStore. As shown, updates to an object are passed through Tapestry to the primary replica for that object. Once the updates are serialized and committed by the primary replica, they are passed down the dissemination tree to secondary replicas that are currently caching the object. These updates are applied to replicas, thereby keeping them up-to-date. Once updates are applied, they become visible to clients sharing that object. Simultaneous with updating secondary replicas, the primary replica encodes new data in an erasure code, sending the resulting fragments to other OceanStore servers for long-term storage.

Note that Figure 2 illustrates the path of updates for a single object. As shown in Section 6, the process of committing updates by the primary replica is computationally intensive. Thus, it is an important aspect of the system that primary replicas can be distributed among inner ring servers to balance load.

3.6 The Primary Replica

Section 3.2 shows that each data object in OceanStore is assigned an inner ring, a set of servers that implement the object's primary replica. These servers securely apply updates and create new versions. They serialize concurrent writes, enforce access control, check update predicates, and sign a heartbeat for each new version.

To construct this primary replica in a fault-tolerant manner, we adapt a Byzantine agreement protocol developed by Castro and Liskov [4]. Byzantine agreement is a distributed decision process in which all non-faulty participants reach the same decision as long as more than two-thirds of the participants follow the protocol correctly. That is, for a group of size $3f + 1$, no more than f servers may be faulty. The faulty machines may fail arbitrarily: they may halt, send incorrect messages, or deliberately try to disrupt the agreement. Unfortunately, Byzantine agreement requires a number of messages quadratic in the number of participants, so it is infeasible for use in synchronizing a large number of replicas; this infeasibility motivates our desire to keep the primary replicas of an object small in number.

The Castro and Liskov algorithm has been shown to perform quite well in a fault-tolerant network file system. We modify the algorithm for our distributed file system in the following important ways.

Public Key Cryptography: Byzantine agreement protocols require that participants authenticate the messages they send. There are two versions of the Castro-Liskov protocol. In the first version, this authentication was accomplished with public-key cryptography. A more recent version used symmetric-key message authentication codes (MACs) for performance reasons: a MAC can be computed two or three orders of magnitude faster than a public-key signature.

MACs, however, have a downside common to all symmetric key cryptography: they only authenticate messages between two fixed machines. Neither machine can subsequently prove the authenticity of a message to a third party. MACs complicate Castro and Liskov's latter algorithm, but they feel the resulting improvement in performance justifies the extra complexity.

In OceanStore we use aggressive replication to improve data object availability and client-perceived access latency. Without third-party verification, each machine would have to communicate directly with the inner ring to validate the integrity of the data it stores. The computation and communication required to keep each replica consistent would limit the maximum number of copies of any data object–even for read-only data.

We therefore modified the Castro-Liskov protocol to use MACs for all communication internal to the inner ring, while using public-key cryptography to communicate with all other machines. In particular, a digital signature certifies each agreement result. As such, secondary replicas can locally verify the authenticity of data received from other replicas or out of the archive. Consequently, most read traffic can be satisfied completely by the second tier of replicas. Even when clients insist on communicating directly with the ring for maximum consistency, it need only provide a heartbeat certifying the latest version; data blocks can still be sourced from secondary replicas.

Computing signatures is expensive; however, we amortize the added cost of each agreement over the number of replicas that receive the result. Public-key cryptography allows the inner ring to push updates to replicas without authenticating the result for each individually. Also, the increased ability of secondary replicas to handle client requests without contacting the inner ring may significantly reduce the number of agreements performed on the inner ring. We analyze the full performance implications of digital signatures in Section 6.

Proactive Threshold Signatures: Traditional Byzantine agreement protocols guarantee correctness if no more than f servers fail *during the life of the system*; this restriction is impractical for a long-lived system. Castro and Liskov address this shortcoming by rebooting servers from a secure operating system image at regular intervals [4]. They assume that keys are protected via cryptographic hardware and that the set of servers participating in the Byzantine agreement is fixed.

In OceanStore, we would like considerable more flexibility in choosing the membership of the inner ring. To do so, we employ *proactive threshold signatures* [22], which allow us to replace machines in the inner ring without changing public keys.

A threshold signature algorithm pairs a single public key with l private *key shares*. Each of the l servers uses its key share to generate a *signature share*, and any k correctly generated signature shares may be combined by any party to produce a full signature. We set $l = 3f + 1$ and $k = f + 1$, so that a correct signature proves that the inner ring made a decision under the Byzantine agreement algorithm.

A proactive threshold signature scheme is a threshold signature scheme in which a new set of l key shares may be computed that are independent of any previous set; while k of the new shares may be combined to produce a correct signature, signature shares generated from key shares from distinct sets cannot be combined to produce full signatures.

To change the composition of an inner ring, the existing hosts of that ring participate in a distributed algorithm with the new hosts to compute a second set of l shares. These shares are independent of the original set: shares from the two sets cannot be combined to produce a valid signature. Once the new shares are generated and distributed to the new servers, the old servers delete their old shares. By the Byzantine assumption, at most $f = k - 1$ of the old servers are faulty, and the remainder will correctly delete their old key shares, rendering it impossible to generate new signatures with the them. Because the public key has not changed, however, clients can still verify new signatures using the same public key.

A final benefit of threshold signatures is revealed when they are combined with the routing and location services of Tapestry. Rather than directly publishing their own GUIDs, the hosts in the inner ring publish themselves under the AGUIDs of the objects they serve. When the composition of the ring changes, the new servers publish themselves in the same manner. Since the ring's public key does not change, clients of the ring need not worry about its exact composition; the knowledge of its key and the presence of Tapestry are sufficient to contact it.

The Responsible Party: Byzantine agreement allows us to build a fault-tolerant primary replica for each data object. By also using public-key cryptography, threshold signatures, and Tapestry, we achieve the ability to dynamically change the hosts implementing that replica in response to failures or changing usage conditions. One difficulty remains, however: who chooses the hosts in the first place?

To solve this problem, we rely on an entity known as the *responsible party*, so named for its responsibility to choose the hosts that make up inner rings. This entity is a server which publishes sets of failure-independent nodes discovered through offline measurement and analysis [37]. Currently, we access this server through Tapestry, but simply publishing such sets on a secure web site would also suffice. An inner ring is created by selecting one node from each of $3f + 1$ independent sets.

Superficially, the responsible party seems to introduce a single point of failure into the system. While this is true to an extent, it is a limited one. The responsible party itself never sees the private key shares used by the primary replica; these are generated through a distributed algorithm involving only the servers of the inner ring, and new groups of shares are also generated in this manner. Thus, a compromise in the privacy of the data stored by the responsible party will not endanger the integrity of file data. As with primary replicas, there can be many responsible parties in the system; the responsible party thus presents no scalability issue. Furthermore, the online interface to the responsible party only provides the read-only results of an offline computation; there are known solutions for building scalable servers to provide such a service.

4 Prototype

This section describes important aspects of the implementation of the prototype, as well as the ways in which it differs from our system description.

4.1 Software Architecture

We built Pond in Java, atop the Staged Event-Driven Architecture (SEDA) [39], since prior research indicates that event-driven servers behave more gracefully under high load than traditional threading mechanisms [39]. Each Pond subsystem is implemented as a *stage*, a self-contained component with its own state and thread pool. Stages communicate with each other by sending *events*.

Figure 3 shows the main stages in Pond and their interconnections. Not all components are required for all OceanStore machines; stages may be added or removed to reconfigure a server. Stages on the left are necessary for servers in the inner ring, while stages on the right are generally associated with clients' machines.

The current code base of Pond contains approximately 50,000 semicolons and is the work of five core graduate student developers and as many undergraduate interns.

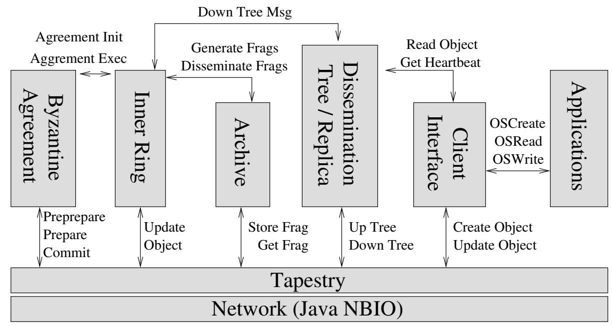

Figure 3: *Prototype Software Architecture.* Pond is built atop SEDA. Components within a single host are implemented as *stages* (shown as boxes) which communicate through events (shown as arrows). Not all stages run on every host; only inner ring hosts run the Byzantine agreement stage, for example.

4.2 Language Choice

We implemented Pond in Java for several reasons. The most important was speed of development. Unlike C or C++, Java is strongly typed and garbage collected. These two features greatly reduce debugging time, especially for a large project with a rapid development pace.

The second reason we chose Java was that we wanted to build our system using an event driven architecture, and the SEDA prototype, SandStorm, was readily available. Furthermore, unlike multithreaded code written in C or C++, multithreaded code in Java is quite easy to port. To illustrate this portability, our code base, which was implemented and tested solely on Debian GNU/Linux workstations, was ported to Windows 2000 in under a week of part-time work.

Unfortunately our choice of programming language also introduced some complications; foremost among these is the unpredictability introduced by garbage collection. All current production Java Virtual Machines (JVMs) we surveyed use so-called "stop the world" collectors, in which every thread in the system is halted while the garbage collector runs[2]. Any requests currently being processed when garbage collection starts are stalled for on the order of one hundred milliseconds. Requests that travel across machines may be stopped by several collections in serial. While this event does not happen often, it can add several seconds of delay to a task normally measured in tens of milliseconds.

To adjust for these anomalies, we report the median value and the 0th and 95th percentile values for experiments that are severely effected by garbage collection instead of the more typical mean and standard deviation. We feel this decision is justified because the effects of garbage collection are merely an artifact of our choice of language rather than an inherent property of the system; an implementation of our system in C or C++ would not exhibit this behavior.

4.3 Inner Ring Issues

Most of the core functionality of the inner ring is implemented in Pond, with the following exception. We do not currently implement view changes or checkpoints, two components of the Castro-Liskov algorithm which are used to handle host failure. However, this deficiency should not sufficiently affect our results; Castro and Liskov found only a 2% performance degradation due to recovery operations while running the Andrew500 benchmark [4] on their system.

Lastly, our current signature scheme is a threshold version of RSA developed by Shoup [32]. We plan to implement a proactive algorithm, most likely Rabin's [22], soon; since the mathematics of the two schemes is similar, we expect similar performance from them as well.

5 Experimental Setup

We use two experimental test beds to measure our system. The first test bed consists of a local cluster of forty-two machines at Berkeley. Each machine in the cluster is a IBM xSeries 330 1U rackmount PC with two 1.0 GHz Pentium III CPUs, 1.5 GB ECC PC133 SDRAM, and

[2]We currently use JDK 1.3 for Linux from IBM. See http://www.ibm.com/developerworks/java/jdk/linux130/.

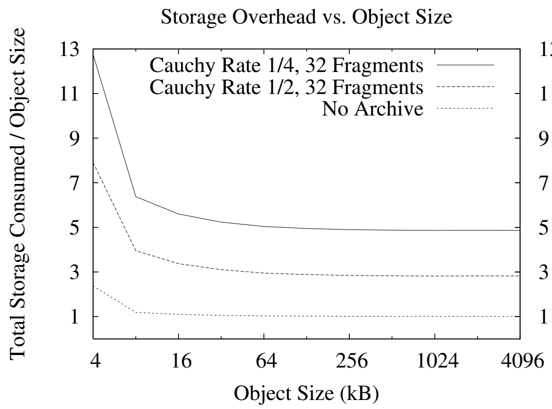

Figure 4: *Storage Overhead.* Objects of size less than the block size of 8 kB still require one block of storage. For sufficiently large objects, the metadata is negligible. The cost added by the archive is a function of the encoding rate. For example, a rate 1/4 code increases the storage cost by a factor of 4.8.

two 36 GB IBM UltraStar 36LZX hard drives. The machines use a single Intel PRO/1000 XF gigabit Ethernet adaptor to connect to a Packet Engines PowerRail gigabit switch. The operating system on each node is Debian GNU/Linux 3.0 (woody), running the Linux 2.4.18 SMP kernel. The two disks run in software RAID 0 (striping) mode using md raidtools-0.90. During our experiments the cluster is otherwise unused.

The second test bed is PlanetLab, an open, global test bed for developing, deploying, and accessing new network services (see http://www.planet-lab.org/). The system currently operates on 101 nodes spread across 43 sites throughout North America, Europe, Australia, and New Zealand. While the hardware configuration of the machines varies slightly, most of the nodes are 1.2 GHz Pentium III CPUs with 1 GB of memory.

For some of our experiments we use a subset of PlanetLab distributed throughout the San Francisco Bay Area in California, USA. The machines that comprise the group of "Bay Area" servers include one machine from each of the following sites: University of California in Berkeley, CA; Lawrence Berkeley National Laboratories in Berkeley, CA; Intel Research Berkeley in Berkeley, CA; and Stanford University in Palo Alto, CA.

6 Results

In this section, we present a detailed performance analysis of Pond. Our results demonstrate the performance characteristics of the system and highlight promising areas for further research.

Key Size	Update Size	Archive	Update Latency (ms) 5%	Median	95%
512	4 kB	off	36	37	38
		on	39	40	41
	2 MB	off	494	513	778
		on	1037	1086	1348
1024	4 kB	off	94	95	96
		on	98	99	100
	2 MB	off	557	572	875
		on	1098	1150	1448

Table 2: *Results of the Latency Microbenchmark in the Local Area.* All nodes are hosted on the cluster. Ping latency between nodes in the cluster is 0.2 ms. We run with the archive enabled and disabled while varying the update size and key length.

6.1 Storage Overhead

We first measure the storage overhead imposed by our data model. As discussed in Section 2, the data object is represented as a B-tree with metadata appended to the top block. When the user data portion of the data object is smaller than the block size, the overhead of the top block dominates the storage overhead. As the user data increases in size, the overhead of the top block and any interior blocks becomes negligible. Figure 4 shows the overhead due to the B-tree for varying data sizes.

The storage overhead is further increased by erasure coding each block. Figure 4 shows that this increase is proportional to the inverse of the rate of encoding. Encoding an 8kB block using a rate $r = \frac{1}{2}$ ($m = 16, n = 32$) and $r = \frac{1}{4}$ ($m = 16, n = 64$) code increases the storage overhead by a factor of 2.7 and 4.8, respectfully. The overhead is somewhat higher than the inverse rate of encoding because some additional space is required to make fragments self-verifying. See [38] for details.

6.2 Update Performance

We use two benchmarks to understand the raw update performance of Pond.

The Latency Microbenchmark: In the first microbenchmark, a single client submits updates of various sizes to a four-node inner ring and measures the time from before the request is signed until the signature over the result is checked. To warm the JVM[3], we update 40 MB of data or perform 1000 updates, depending on the size of the update being tested. We pause for ten seconds to allow the system to quiesce and then perform a number of updates, pausing 100 ms between the response from one update and the request for the next. We report

[3]Because Java code is generally optimized at runtime, the first several executions of a line of code are generally slow, as the runtime system is still optimizing it. Performing several passes through the code to allow this optimization to occur is called *warming* the JVM.

Phase	Time (ms) 4 kB Update	2 MB Update
Check Validity	0.3	0.4
Serialize	6.1	26.6
Update	1.5	113.0
Archive	4.5	566.9
Sign Result	77.8	75.8

Table 3: *Latency Breakdown of an Update.* The majority of the time in a small update performed on the cluster is spent computing the threshold signature share over the result. With larger updates, the time to apply and archive the update dominates signature time.

Inner Ring	Client	Avg. Ping	Update Size	Update Latency (ms) 5%	Median	95%
Cluster	Cluster	0.2	4 kB	98	99	100
			2 MB	1098	1150	1448
Cluster	UCSD	27.0	4 kB	125	126	128
			2 MB	2748	2800	3036
Bay Area	UCSD	23.2	4 kB	144	155	166
			2 MB	8763	9626	10231

Table 4: *Results of the Latency Microbenchmark Run in the Wide Area.* All tests were run with the archive enabled using 1024-bit keys. “Avg. Ping” is the average ping time in milliseconds from the client machine to each of the inner ring servers. UCSD is the University of California at San Diego.

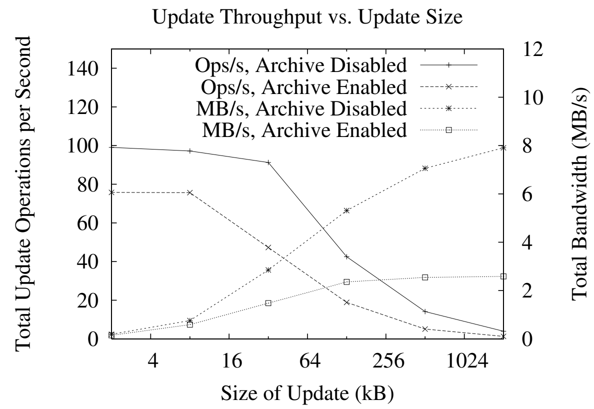

Figure 5: *Throughput in the Local Area.* This graph shows the update throughput in terms of both operations per second (left axis) and bytes per second (right axis) as a function of update size. While the ops/s number falls off quickly with update size, throughput in bytes per second continues to increase. All experiments are run with 1024-bit keys. The data shown is the average of three trials, and the standard deviation for all points is less than 3% of the mean.

the latency of the median, the fifth percentile, and the ninety-fifth percentile.

We run this benchmark with a variety of parameters, placing the nodes in various locations through the network. Table 2 presents the results of several experiments running the benchmark on the cluster, which show the performance of the system apart from wide-area network effects. This isolation highlights the computational cost of an update. While 512-bit RSA keys do not provide sufficient security, we present the latency of the system using them as an estimate of the effect of increasing processor performance. Signature computation time is quadratic in the number of bits; a 1024-bit key signature takes four times as long to compute as a 512-bit one. The performance of the system using 512-bit keys is thus a conservative estimate of its speed after two iterations of Moore’s law (roughly 36 months).

Table 3 presents a breakdown of the latency of an update on the cluster. In the check validity phase, the client’s signature over the object is checked. In the serialization phase, the inner ring servers perform the first half of the Byzantine agreement process, ensuring they all process the same updates in the same order. In the update and archive phases, the update is applied to a data object and the resulting version is archived. The final phase completes the process, producing a signed heartbeat over the new version. It is clear from Table 3 that most of the time in a small update is spent computing the threshold signature share over the result. With larger updates, the time to apply and archive the update is large, and the signature time is less important. Although we have not yet quantified the cost of increasing the ring size, the serialize phase requires quadratic communication costs in the size of the ring. The other phases, in contrast, scale at worst linearly in the ring size.

Table 4 presents the cost of the update including network effects. Comparing rows one and two, we see that moving the client to UCSD adds only the network latency between it and the inner ring to the total update time for small updates. Comparing rows two and three we see that distributing the inner ring throughout the Bay Area increases the median latency by only 23% for small updates. Since increased geographic scale yields increased failure independence, this point is very encouraging. For larger updates, bandwidth limitations between the PlanetLab machines prevent optimal times in the wide area; it is thus important that a service provider implementing a distributed inner ring supply sufficient bandwidth between sites.

The Throughput Microbenchmark: In the second microbenchmark, a number of clients submit updates of various sizes to a four-node inner ring. Each client submits updates for a different data object. The clients create their objects, synchronize themselves, and then update the object as many times as possible in a 100 second period. We measure the number of updates completed by all clients and report the update and data throughput.

Figure 5 shows the results of running the throughput

IR Location	Client Location	Throughput (MB/s)
Cluster	Cluster	2.59
Cluster	PlanetLab	1.22
Bay Area	PlanetLab	1.19

Table 5: *Throughput in the Wide Area.* The throughput for a distributed ring is limited by the wide-area bandwidth. All tests are run with the archive on and 1024-bit keys.

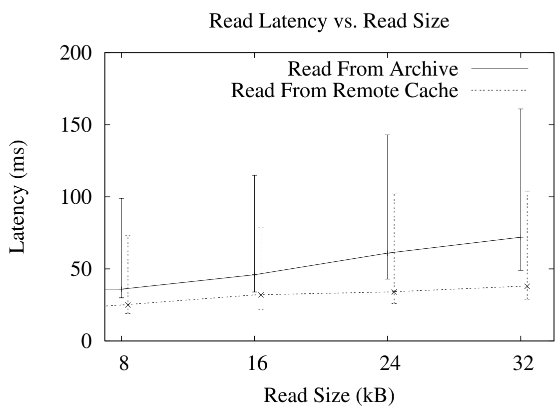

Figure 6: *Latency to Read Objects from the Archive.* The latency to read data from the archive depends on the latency to retrieve enough fragments for reconstruction.

test on the cluster. Again, running the test in the local area illustrates the computational limitations of the inner ring. Lines sloping downward show the number of operations completed per second as a function of the update size and archival policy; lines sloping upward show the corresponding throughput in megabytes per second.

If the inner ring agrees about each update individually, the maximum possible number of operations completed per second is bounded by the speed of threshold signature generation, or approximately 10 operations per second. Instead, the inner ring batches updates and agrees on them in groups (as suggested by [4]); because of this, we have found that the throughput of the system does not change much when using 512-bit keys. Unfortunately, there are other costs associated with each update, so batching only helps to a degree. As suggested by Table 3, however, as the update size increases the signature phase becomes only a small part of the load, so throughput in megabytes per second continues to increase. From Figure 5, we see the maximum throughput of the prototype with the archive disabled is roughly 8 MB/s.

The throughput of the prototype with the archival subsystem enabled is significantly lower. This is not surprising given the effect of the computationally-intensive archiving process we observed in Table 2. From Figure 5, we see that the maximum sustainable throughput of the archival process is roughly 2.6 MB/s. As such, we plan to focus a significant component of our future work on tuning the archival process.

Table 5 shows the results of running the throughput test with the archive running and hosts located throughout the network. In the wide area, throughput is limited by the bandwidth available.

6.3 Archive Retrieval Performance

To read a data object in OceanStore, a client can locate a replica in Tapestry. If no replica exists, one must be reconstructed from archival fragments. The latency of accessing a replica is simply the latency of through Tapestry. Reconstructing data from the archive is a more complicated operation that requires retrieving several fragments through Tapestry and recomputing the data from them.

To measure the latency of reading data from the archive, we perform a simple experiment. First, we populate the archive by submitting updates of various sizes to a four-node inner ring. Next, we delete all copies of the data in its reconstructed form. Finally, a single client submits disjoint read events synchronously, measuring the time from each request until a response is received. We perform 1,000 reads to warm the JVM, pause for thirty seconds, then perform 1,000 more, with 5 ms between the response to each read and the subsequent request. For comparison, we also measure the cost of reading remote replicas through Tapestry. We report the minimum, median, and 95th percentile latency.

Figure 6 presents the latency of reading objects from the archive running on the cluster. The archive is using a rate $r = \frac{m}{n} = \frac{16}{32}$ code; the system must retrieve 16 fragments to reconstruct a block from the archive. The graph shows that the time to read an object increases with the number of 8kB blocks that must be retrieved. The median cost of reading an object from the archive is never more the 1.7 times the cost of reading from a previously reconstructed remote replica.

6.4 Secondary Replication

In this section, we describe two benchmarks designed to evaluate the efficiency and performance of the dissemination tree that connects the second tier of replicas.

The Stream Benchmark: The first benchmark measures the network resources consumed by streaming data through the dissemination tree from a content creator to a number of replicas. We define the efficiency of the tree as the percentage of bytes sent down high-latency links while distributing an update to every replica. We assume that most high-latency links will either have low bandwidth or high contention; local, low-latency links should be used whenever possible.

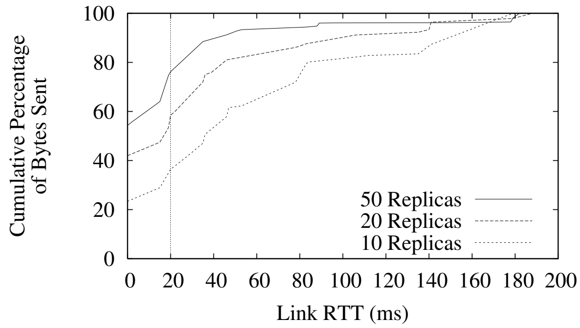

Figure 7: *Results of the Stream Benchmark.* The graph shows the percentage of bytes sent over links of different latency as the number of replicas varies.

In this benchmark, we create a Tapestry network with 500 virtual OceanStore nodes spread across the hosts in 30 PlanetLab sites. We then create a single shared OceanStore data object with a Bay Area inner ring and a variable number of replicas hosted on the seven largest PlanetLab sites. One of these sites lies in the United Kingdom; the other six are in the United States. A single replica repeatedly submits updates that append data to the object. We measure the bandwidth consumed pushing the updates to all other replicas.

Figure 7 shows the percentage of bytes sent across network links of various latencies in this benchmark. According to our metric, the dissemination tree distributes the data efficiently. With only 10 replicas, there are 1.4 replicas per site on average, and 64% of all bytes sent are transmitted across links of latency greater than 20 ms. With 50 replicas, however, there are an average of 7.1 replicas per site, and only 24% of all bytes are sent across links of latency greater than 20 ms.

The Tag Benchmark: The next benchmark measures data sharing in a more interactive scenario, such as a chat room. We arrange for a group of OceanStore replicas to play a distributed game of "tag". To play tag, replicas pass a small piece of data—or *token*—among the group and measure how quickly the token is passed.

In this benchmark, we create a Tapestry network with 200 virtual OceanStore nodes spread across the same 30 PlanetLab sites used in the stream benchmark. We create a single shared data object with a Bay Area inner ring and 50 replicas hosted on the large PlanetLab sites. To pass the token, the replica holding it writes the name of another replica into the data object. A replica receives the token when it reads the new version and finds its name. We measure the average latency over 500 passes.

To put these latencies in perspective, we run two control experiments without using Pond. In these experiments, a coordinator node is placed on one of the machines that hosted an inner ring node in the OceanStore experiment. To pass the token, a replica sends a message to the coordinator; the coordinator forwards the token to the next recipient. In one control experiment, Tapestry is used to communicate between nodes; in the other, TCP/IP is used.

Tokens Passed Using	Latency per Tag (ms)
OceanStore	329
Tapestry	104
TCP/IP	73

Table 6: *Results of the Tag Microbenchmark.* Each experiment was run at least three times, and the standard deviation across experiments was less than 10% of the mean. All experiments are run using 1024-bit keys and with the archive disabled.

As demonstrated by the stream benchmark, the dissemination tree is bandwidth efficient; the tag benchmark shows that this efficiency comes at the cost of latency. Table 6 presents the results of the tag benchmark. In the control cases, the average time to pass the token is 73 ms or 104 ms, depending on whether TCP/IP or Tapestry is used. Using OceanStore, passing the token requires an average of 329 ms. Subtracting the minimum time to perform an update (99 ms, according to Table 4), we see that the latency to pass the token through the dissemination tree is 2.2 times slower than passing the token through Tapestry and 3.2 times slower than using TCP/IP.

6.5 The Andrew Benchmark

To illustrate the performance of Pond on a workload familiar to systems researchers, we implemented a UNIX file system interface to OceanStore using an NFS loopback server [19] and ran the Andrew benchmark. To map the NFS interface to OceanStore, we store files and directories as OceanStore data objects. We use a file's AGUID as its NFS file handle; directories are represented as simple lists of the files that they contain. The information normally stored in a file's inode is stored in the metadata portion of the OceanStore object.

When an application references a file, the replica code creates a local replica and integrates itself into the corresponding object's dissemination tree. From that point on, all changes to the object will be proactively pushed to the client down the dissemination tree, so there is no need to consult the inner ring on read-only operations.

Write operations are always sent directly to the inner ring. NFS semantics require that client writes not be comingled, but imposes no ordering between them. The inner ring applies all updates atomically, so enclosing each write operation in a single update is sufficient to satisfy the specification; writes never abort. Directories must be handled more carefully. On every directory

	LAN			WAN		
	Linux	OceanStore		Linux	OceanStore	
Phase	NFS	512	1024	NFS	512	1024
I	0.0	1.9	4.3	0.9	2.8	6.6
II	0.3	11.0	24.0	9.4	16.8	40.4
III	1.1	1.8	1.9	8.3	1.8	1.9
IV	0.5	1.5	1.6	6.9	1.5	1.5
V	2.6	21.0	42.2	21.5	32.0	70.0
Total	4.5	37.2	73.9	47.0	54.9	120.3

Table 7: *Results of the Andrew Benchmark.* All experiments are run with the archive disabled using 512 or 1024-bit keys, as indicated by the column headers. Times are in seconds, and each data point is an average over at least three trials. The standard deviation for all points was less than 7.5% of the mean.

change, we specify that the change only be applied if the directory has not changed since we last read it. This policy could theoretically lead to livelock, but we expect contention of directory modifications by users to be rare.

The benchmark results are shown in Table 7. In the LAN case, the Linux NFS server and the OceanStore inner ring run on our local cluster. In the WAN case, the Linux NFS server runs on the University of Washington PlanetLab site, while the inner ring runs on the UCB, Stanford, Intel Berkeley, and UW sites. As predicted by the microbenchmarks, OceanStore outperforms NFS in the wide area by a factor of 4.6 during the read-intensive phases (III and IV) of the benchmark. Conversely, the write performance (phases I and II) is worse by as much as a factor of 7.3. This latter difference is due largely to the threshold signature operation rather than wide-area latencies; with 512-bit keys, OceanStore is no more than a factor of 3.1 slower than NFS. When writes are interspersed with reads and computation (phase V), OceanStore performs within a factor of 3.3 of NFS, even with large keys.

7 Related Work

A number of distributed storage systems have preceded OceanStore; notable examples include [31, 13, 8]. More recently, as the unreliability of hosts in a distributed setting has been studied, Byzantine fault-tolerant services have become popular. FarSite [3] aims to build an enterprise-scale distributed file system, using Byzantine fault-tolerance for directories only. The ITTC project [40] and the COCA project [42] both build certificate authorities (CAs) using threshold signatures; the later combines this scheme with a quorum-based Byzantine fault-tolerant algorithm. The Fleet [16] persistent object system also uses a quorum-based algorithm.

Quorum-based Byzantine agreement requires less communication per replica than the state-machine based agreement used in OceanStore; however, it tolerates proportionally less faults. It was this tradeoff that led us to our architecture; we use primary-copy replication [10] to reduce communication costs, but implement the primary replica as a small set servers using state-machine Byzantine agreement to achieve fault tolerance.

In the same way that OceanStore is built atop Tapestry, a number of other peer-to-peer systems are constructing self-organizing storage on distributed routing protocols. The PAST project [28] is producing a global-scale storage system data using replication for durability. The cooperative file system (CFS) [7] also targets wide-area storage. We chose Tapestry for its locality properties; functionally, however, other routing protocols ([17, 18, 23, 27, 33]) could be used instead. Like OceanStore, both PAST and CFS provide probabilistic guarantees of performance and robustness; unlike OceanStore, however, they are not designed for write sharing. Ivy [21] is a fully peer-to-peer read-write file system built atop the CFS storage layer. Unlike OceanStore, it provides no single point of consistency for data objects; conflicting writes must be repaired at the application level. Similarly, Pangaea [29] provides only "eventual" consistency in the presence of conflicting writes. By supporting Bayou-style update semantics and having a single point of consistency per object, OceanStore is able to support higher degrees of consistency (including full ACID semantics) than Ivy or Pangaea; by distributing this single point through a Byzantine agreement protocol, OceanStore avoids losses of availability due to server failures.

Still other distributed storage systems use erasure-coding for durability. One of the earliest is Intermemory [9], a large-scale, distributed system that provides durable archival storage using erasure codes. The Pasis [41] system uses erasure-coding to provide durability and confidentiality in a distributed storage system. Pasis and Intermemory both focus on archival storage, rather than consistent write sharing. Mnemosyne [11] combines erasure-codes with client-directed refresh to achieve durability; clients rewrite data at a rate sufficient to guarantee the desired survival probability.

A final class of systems are also related to OceanStore by the techniques they use, if not in the focus of their design. Publius [15], the Freenet [6], and Eternity Service [1] all focus on preventing censorship of distributed data, each in their own way. Publius uses threshold cryptography to allow a host to store data without knowing its content, as a method of allowing deniability for the host's operators. Freenet also uses coding for deniability, and is built on a routing overlay similar in interface to Tapestry. Finally, the Eternity Service uses erasure coding to make censoring data beyond the resources of any one entity.

8 Conclusions and Future Work

We have described and characterized Pond, the OceanStore prototype. While many important challenges remain, this prototype is a working subset of the vision presented in the original OceanStore paper [14].

Building this prototype has refined our plans for future research. We initially feared that the increased latency of a distributed Byzantine agreement process might be prohibitive, a fear this work has relieved. Instead, threshold signatures have proven far more costly than we anticipated, requiring an order of magnitude more time to compute than regular public key signatures. We plan to spend significant time researching more efficient threshold schemes, or possibly even alternate methods for achieving the benefits they provide. Likewise, we plan to focus on improving the speed of generating erasure-encoded fragments of archival data. Not discussed in this work is the overhead of virtualization. While the latency overhead of Tapestry has been examined before [24], quantifying the additional storage costs it imposes is a topic for future research.

Our future work should not focus entirely on performance, however. One interesting property of the current system is the self-maintaining algorithms it employs. Tapestry automatically builds an overlay network that efficiently finds network resources, and the dissemination tree self-organizes to keep replicas synchronized. The use of threshold signatures allows the inner ring to change its composition without affecting the rest of the system. We hope to make more aspects of the system self-maintaining in the future. For example, algorithms for predictive replica placement and efficient detection and repair of lost data [35] are vital for lowering the management costs of distributed storage systems like OceanStore.

Increased stability and fault-tolerance are also important if Pond is to become a research vehicle for even more interesting projects. Our work in benchmarking Tapestry and its peers [25] was started with the intention of improving the stability of the lowest layer of Pond. Moreover, network partitions are a problem for most overlay networks, and further research is needed to study the behavior of Tapestry under partition. As the stability of Tapestry improves, our focus will shift to higher layers of the system.

Finally, the OceanStore data model has proven expressive enough to support several interesting applications, including a UNIX file system with time travel, a distributed web cache, and an email application. Nonetheless, the developement of these applications has pointed out areas in which the OceanStore API could be improved; a more intuitive API will hopefully spur the developement of further OceanStore applications.

9 Availability

The Pond source code and benchmarks are published under the BSD license and are freely available from http://oceanstore.cs.berkeley.edu.

10 Acknowledgements

We would like thank IBM for providing the hardware in our cluster, and all of the groups that have contributed to PlanetLab. Without these two testbeds, this work would not have been possible. Brent Chun and Mike Howard were particularly helpful with our experiments. Anthony Joseph and Timothy Roscoe provided valuable input on the design and implementation of Pond. Jeremy Stribling's debugging skills were instrumental in bringing up large Tapestry networks. Finally, we are grateful to our anonymous reviewers and Frans Kaashoek, our paper's steward, whose comments and advice have greatly improved this work.

References

[1] R. Anderson. The eternity service. In *Proceedings of Pragocrypt*, 1996.

[2] J. Bloemer et al. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, The International Computer Science Institute, Berkeley, CA, 1995.

[3] 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.

[4] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proc. of OSDI*, 2000.

[5] Y. Chen, R. Katz, and J. Kubiatowicz. SCAN: A dynamic, scalable, and efficient content distribution network. In *Proc. of International Conference on Pervasive Computing*, 2002.

[6] I. Clark, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, Berkeley, CA, July 2000.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, October 2001.

[8] A. Demers et al. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of IEEE Workshop on Mobile Computing Systems & Applications*, 1994.

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

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of ACM SIGMOD Conf.*, June 1996.

[11] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Proc. of IPTPS*, March 2002.

[12] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proc. of ACM SPAA*, pages 41–52, August 2002.

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

[14] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[15] A. Rubin M. Waldman and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, 2000.

[16] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *DISCEX II*, 2001.

[17] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc. of ACM PODC Symp.*, 2002.

[18] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of IPTPS*, 2002.

[19] D. Mazières. A toolkit for user-level file systems. In *Proc. of USENIX Summer Technical Conf.*, June 2001.

[20] R. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, pages 369–378. Springer-Verlag, 1988.

[21] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI*, 2002.

[22] T. Rabin. A simplified approach to threshold and proactive RSA. In *Proceedings of Crypto*, 1998.

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

[24] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. of INFOCOM*. IEEE, June 2002.

[25] S. Rhea, T. Roscoe, and J. Kubiatowicz. DHTs need application-driven benchmarks. In *Proc. of IPTPS*, 2003.

[26] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance free global storage in oceanstore. In *Proc. of IEEE Internet Computing*. IEEE, September 2001.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, November 2001.

[28] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.

[29] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of OSDI*, 2002.

[30] 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*, December 1999.

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

[32] V. Shoup. Practical threshold signatures. In *Proc. of EUROCRYPT*, 2000.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*. ACM, August 2001.

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

[35] H. Weatherspoon and J. Kubiatowicz. Efficient heartbeats and repair of softstate in decentralized object location and routing systems. In *Proc. of SIGOPS European Workshop*, 2002.

[36] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, March 2002.

[37] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. of International Workshop on Reliable Peer-to-Peer Distributed Systems*, October 2002.

[38] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Proc. of International Workshop on Future Directions of Distributed Systems*, 2002.

[39] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of ACM SOSP*, October 2001.

[40] T. Wu, M. Malkin, and D. Boneh. Building intrusion-tolerant applications. In *Proc. of USENIX Security Symp.*, August 1999.

[41] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, August 2000.

[42] L. Zhou, F. Schneider, and R. van Renesse. Coca: A secure distributed on-line certification authority. Technical Report 2000-1828, Department of Computer Science, Cornell University, Ithaca, NY USA, 2000.