Abstract

OceanStore is a persistent data store designed to achieve global scale. We present explicit details of the first OceanStore prototype, called Pond. This prototype contains many of the features that we envision for 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. We discuss the details of the prototype, illustrate the performance of individual components through microbenchmarks, and analyze the performance of the complete system via the Andrew benchmark. We finish by illustrating the performance of the prototype running on servers spread throughout the United States.

1 Introduction

Exponential growth in processor performance, storage capacity, and network bandwidth have fundamentally changed the way in which we design systems. Only recently have we been able to envision integrated computer systems composed of millions of components spread across geographical distances. The advantages of global-scale systems lie in their ubiquity—pieces of the system are geographically close to users regardless of where they reside. Such systems also derive significant advantages from redundancy, since there are many equivalent components and communication paths.

One compelling use for global-scale systems is information storage and retrieval. This paper explores the design and implementation of the OceanStore prototype [K+00, RWE+01], a distributed storage repository designed to incorporate millions of servers and exabytes of data. To users, OceanStore presents the illusion of a large, secure, virtual disk.

The OceanStore design is based around a set of simple properties. First among these is universal access of information: data should be highly available from anywhere within the network, and access to it should be unimpeded by physical or administrative boundaries. Next, the system should balance the complementary properties of privacy and flexible information sharing: the choice of whether to share or protect data should reside with its owner, and cryptography should be used to enforce that decision. Also complementary to privacy is integrity: the system should guarantee that data can be verified to be correct and from a well-defined source. A crucial component of flexible sharing is that the consistency model must be easily understandable and usable. Finally, the system should provide data durability, guaranteeing that data is not lost during exceptional failures.

In addition to these goals, the OceanStore design is fundamentally impacted by two axioms. First, the infrastructure is untrusted except in aggregate. Servers and routers may fail at arbitrary times, users or servers may be malicious, and data entered into the infrastructure is subject to snooping by third parties. Consequently, one of the key design goals of OceanStore is isolation against faults and the assumption of untrustworthiness.

A second axiom is that the infrastructure is constantly changing. Performance of existing servers and communication paths is not constant, and resources continually enter and exit the network without warning. This leads to the design goal of adaptability: the system must be self-organizing, self-repairing and self-tuning. Adaptability requires built-in redundancy in each component (servers and network paths), as well as dynamic algorithms to efficiently utilize this redundancy. Adaptability brings the additional benefit of minimizing reliance on human intervention, an important constraint in a system with millions of elements.

In this paper, we present explicit details of the
Figure 1: A data object is a sequence of versions, referred to by an AGUID. Each version contains metadata (denoted by $M$), blocks of data (denoted by $d_j$), and references to earlier versions. Object structure is described further in Section 3.1.

OceanStore prototype, called Pond. This prototype contains many of the features that we envision for a complete system, including location-independent routing, Byzantine update commitment, push-based update of cached copies through an overlay multicast network, and continuous data archiving to erasure-coded form. We describe the system in the following three sections. Then, Sections 5 and 6 discuss our experimental framework and analyze the performance of the complete system via micro- and macro-benchmarks. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 System Overview

This section describes OceanStore at a high level. We present the principle system components and highlight important design decisions.

2.1 Versioning

Figure 1 illustrates an OceanStore data object, the analog to a file in a traditional file system. A data object is named by an active GUID (AGUID), which is a cryptographically-secure hash of the concatenation of an application-specified name and the owner’s public key.

A data object is an ordered sequence of read-only versions. In principle, every version of every object is kept forever. Each version contains metadata, an array of bytes, and one or more references to previous versions. Each version is named by a secure hash of its contents, called its VGUID. This naming mechanism allows clients to verify that they have correctly received the data that they requested.

2.2 Replication

OceanStore replicates data objects to increase performance and availability. Each object is stored on several machines, and clients may read from any of them. Replication allows clients to access their data despite individual machine failures or network outages.

Unfortunately, simple replication complicates trust and consistency models. Consequently, we implement primary-copy replication [GHOS96]. Each object has a single primary replica, which 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. The primary replica also enforces access control restrictions and serializes concurrent updates from multiple users.

We implement the primary replica as a small set of cooperating servers, because no single machine can be trusted with this amount of control over a user’s data. These servers, called the inner ring, use a Byzantine-fault-tolerant protocol to agree on all updates to the data object and sign the result. This protocol allows the ring to operate correctly even if some members fail or behave maliciously.

All other copies of the data object, known as secondary replicas, are simply soft-state copies that stay more or less synchronized with the primary replica. This second tier of replication reduces the load on the inner ring caused by read traffic. Secondary replicas are placed near or on client machines, providing fast access and reducing sensitivity to network outages.

2.3 Archival Storage

OceanStore provides an extremely high level of durability by placing every version of every object into an archival storage subsystem. New versions are archived as soon as they are created. We examine the impact of this choice later, in Section 3.4.

The archival subsystem fragments encodes the data using erasure codes [BKK+95] and calculates secure hashes over the encoded data. These hashes both name and verify both the data and encoded fragments [WWK02]. The archive then distributes the fragments to machines with low probability of simultaneous failures.

Erasure codes ensure that the original data can be reconstructed as long as a fraction of the fragments are...
available. For example, a code that doubles the size of the data produces several fragments, any half of which are sufficient to recover the original data. OceanStore distributes enough fragments to guarantee that only a system-wide catastrophe could destroy a single version of any file.

2.4 Location-Independent Routing

OceanStore uses a distributed object location and routing system (DOLR) called Tapestry [ZJK01]. Tapestry exports a simple interface through which clients forward messages to objects and blocks by name – regardless of their location.

Tapestry is built as an overlay network that scales to large numbers of objects and machines. This scalability is key to the overall scalability of OceanStore. Tapestry also automatically adapts to the insertion of new nodes and the (possibly unplanned) removal of failed nodes [HKRZ02].

Each replica (block, fragment) is published in Tapestry using its AGUID (VGUID, BGUID) as a label. Similarly, client requests for an object are addressed with the same label. If several machines publish the same label, Tapestry will usually forward the request to one that is nearby in the underlying topology. In the worst case, routing requires a number of overlay hops that is logarithmic in the size of the overlay network.

2.5 Putting it All Together

Now that we have briefly described the main components of OceanStore we can demonstrate how they interact. Figure 2 illustrates the path of an update and delineates the important latencies in the system.

3 System Description

In this section we describe four aspects of the OceanStore implementation in greater detail. We focus on the aspects that most affect system performance.

3.1 Self-Verifying Data Model

To simultaneously provide versioning support and random access to data, a version is stored in a data structure similar to a B-tree. The most notable feature of the B-tree is that a block references a child by its block GUID (BGUID), a secure hash over the contents of a block. The VGUID is the same as the BGUID of the top block. The current implementation of the B-tree places all data in blocks of fixed size; an implementation that supports data blocks of arbitrary size would allow more efficient data manipulation [MCM01].

BGUIDs are created by the archival subsystem during the process of fragmentation, as described in Section 3.4. By design, a BGUID verifies the contents of the block it names (including references to any children) in a cryptographically-secure manner. The VGUID, then, is a cryptographically-secure, self-verifying key for the whole version.

An update adds a new version to the chain of versions in a data object. Updates are represented as an array of potential actions each guarded by a predicate. Only the action guarded by the first predicate that evaluates to true is applied. Because only the inner ring can produce valid signatures over new versions, the inner ring assures consistency of the object.

Because versions and their composite blocks are immutable, the B-tree can implement copy-on-write by referring to blocks from previous versions. In this manner, small updates create a proportionally small number of new blocks. Returning to Figure 1, in version $i + 1$, only data blocks $d_i^6$ and $d_i^7$ were changed from version $i$. Version $i + 1$ references blocks in the previous version for all unchanged blocks.

3.2 The Inner Ring

Each data object is assigned an inner ring, a set of servers that together implement the object’s primary replica. These servers securely apply updates and cre-
ate new versions. They serialize concurrent writes, enforce access control, check update predicates, and sign a heartbeat for each new version.

The inner ring maintains consistency among its servers using a Byzantine agreement protocol developed by Castro and Liskov [CL99]. 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.

The Castro and Liskov algorithm performs well in a fault-tolerant file system. We modify the algorithm for OceanStore in the following two important ways.

Public Key Cryptography: Byzantine agreement protocols require that the participants authenticate the messages they send. In the Castro-Liskov protocol, this authentication is accomplished through symmetric-key message authentication codes (MACs), for performance reasons: a MAC can be computed between two and three orders of magnitude faster than a public-key signature.

MACs have a downside, common to all symmetric key cryptography, in that they only authenticate messages between two machines. Neither machine can later prove the authenticity of a message to a third party. This complicates some portions of Castro and Liskov’s algorithm, but they view the resulting improvement in performance as sufficient justification for 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 include a digital signature over the agreement result. With public-key cryptography, replicas can 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. Also, the inner ring can push updates to replicas without authenticating the result for each individually.

Computing signatures is expensive; however, we can amortize the added cost of each agreement over the number of replicas that receive the result. 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 [CL00]. 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. An entity known as the responsible party dynamically chooses the inner ring servers so as to maximize failure independence. It also adapts the makeup of the inner ring over time to adjust for faults or to enhance locality. Since we would like the ability to change the set of servers participating in an inner ring, the Castro and Liskov solution is infeasible. Instead, we utilize proactive threshold signatures 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 group of $l$ key shares may be generated such that any $k$ of them may be combined to produce a correct signature, but signature shares produced with keys from the previous set cannot be combined with those from the newer set to generate a signature. Because the public key does not change, old certificates need not be purged from the system.

Using a proactive scheme, the responsible party generates a new set of key shares whenever it changes the makeup of an inner ring. It distributes the new shares to the new servers and instructs the old servers to stop
producing signatures. By the Byzantine assumption, at most $f$ of the old servers are faulty, and the remainder will correctly delete their old key shares.

### 3.3 The Secondary Replicas

To reduce read latency and inner ring load, OceanStore aggressively caches objects throughout the infrastructure. While secondary replicas do not participate in the update process, they can satisfy a client’s read request with data that can be verified for authenticity.

The secondary replicas of an object organize automatically into a multicast dissemination tree rooted at the primary replicas. To aid in the creation of trees, replicas are published in Tapestry by AGUID. Requests to join a tree are routed by Tapestry to nearby replicas. This simple approach generally produces good multicast trees, as we will show later.

When a client machine first accesses a data object, a replica is automatically created. This local replica absorbs most of the read traffic locally, reducing access latency and network traffic. The client can receive notification that the object was modified by another client because the primary replicas push the results of the update down the tree. The result can take the form of either a invalidation message or a update message. Additional replica can be created in the network to promote more efficient dissemination tree and to provide proxy support for weak clients.

Because secondary replicas are soft state and provide only best-effort service, clients can optionally specify additional constraints placed on the data provided by the replica. These constraints take the form of predicates that are included in read requests. For example, a predicate may specify that data must be no older than 30 seconds. If a secondary replica cannot satisfy a client’s request locally, it may pass the request to its parent in the dissemination or request an object from the archive.

### 3.4 Archival Subsystem

In this section, we describe the archival architecture and policy decisions made by OceanStore. Each policy has an impact on the archive metrics of durability, space efficiency, bandwidth efficiency, and performance.

**Erasure Coding:** OceanStore utilizes erasure coding to increase the efficiency of storage in the archive. An erasure code provides redundancy without the overhead of strict replication. Erasure codes divide an object into $m$ fragments and recode them into $n$ fragments, where $n > m$. We call $r = \frac{m}{n} < 1$ 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, using a $r = \frac{1}{4}$ encoding on a block divides the block into $m = 16$ fragments and encodes the original $m$ fragments into $n = 64$ fragments; increasing the storage cost by a factor of four.

Systems utilizing erasure codes exhibit a mean time to failure many orders of magnitude higher than replicated systems with similar storage and bandwidth requirements [WK02]. Further, erasure-resilient systems use an order of magnitude less bandwidth and storage to provide similar system durability as replicated systems.

There are many different types of erasure codes with varying performance tradeoffs. In the prototype, we utilize a Cauchy Reed-Solomon code [BKK+95]. Although encoding time scales quadratically with the number of fragments, this is negligible since the algorithm is efficient and we utilize a relatively small number of fragments ($m = 16, n = 32$).

**BGUID Generation:** Each time the archive is invoked by the inner ring, it generates fragments and verification hashes over the fragments to produce BGUIDs [WWK02]. A BGUID enables each fragment and block of the data object to be self-verified by any component in the system. To support delayed-archiving, the inner ring generates a hash over the data, independent of the archive. This verification hash (VHASH) is used to verify cached copies of blocks by secondary
replicas and clients until the archive is run the the
BGUID is generated.

Archival Policy: Traditionally, archiving is performed
infrequently and in the background. We investigate
both delaying and inlining the archival process. inlining-archiving is when the archive is run before returning the
result to the client. This has the benefit that a write time
of vulnerability is the smallest (i.e. more durable), but
user-perceived performance is lower. delayed-archiving
is the more traditional mode, and has the better user-
perceived latency, but writes are vulnerable longer (i.e.
less durable). Both policies are explored in Section 6
against a base case of no-archiving.

4 Prototype

This section describes important aspects of the imple-
mentation 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 Matt Welsh’s Staged Event-
Driven Architecture (SEDA) [WCB01]. Each Pond sub-
system is implemented as a stage, a self-contained com-
ponent with its own state and thread pool. Stages com-
municate with each other by sending events.

Prior research at Berkeley indicates that event-driven
servers behave more gracefully under high load than tra-
ditional threading mechanisms [WCB01]. Our through-
put measurements presented in Section 6 lend support
to this hypothesis.

Figure 4 shows the main stages in Pond and their in-
terconnections. 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.

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.
In our experience, these two features greatly reduce de-
bugging time, especially for a large project with a rapid
development pace.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Languages</th>
<th>Signing Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java alone</td>
<td>Java</td>
<td>45.95 (0.10)</td>
</tr>
<tr>
<td>Java + Libcrypto</td>
<td>Java, C</td>
<td>35.62 (1.68)</td>
</tr>
<tr>
<td>Java + Libgmp</td>
<td>Java, C, Assembly</td>
<td>17.27 (0.03)</td>
</tr>
</tbody>
</table>

Table 1: The average and standard deviation of the time to
generate a single signature share under several implementa-
tions of Shoup’s threshold RSA algorithm.

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. Fur-
thermore, 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\(^1\). 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 col-
lections in serial. While this event does not happen of-
ten, it can add several seconds of delay to a task nor-
maually measured in tens of milliseconds.

To adjust for these anomalies, we report the median
value and the 0th and 95th percentile values for exper-
iments that are severely effected by garbage collection
instead of the more typical mean and standard devia-
tion. We feel this decision is justified because the ef-
fects 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. Also, new de-
velopments in incremental and concurrent garbage col-
collectors should correct this problem when they become
commercially available [BAL+01].

4.3 Inner ring issues

Most of the core functionality of the inner ring is im-
plemented in Pond, with the following exceptions. We
do not currently implement view changes or check-
points, two components of the Castro-Liskov algorithm

\(^1\) We currently use JDK 1.3 for Linux from IBM. See
http://www.ibm.com/developerworks/java/jdk/linux130/
which are used during failure conditions. Our results are thus somewhat optimistic, in that they present our performance in the common case when no failures are present. Moreover, checkpoints require additional computation and network traffic during non-failure modes. We do not expect our results to change drastically once these features are implemented, since the majority of the state that must be checkpointed is already stored in the archive.

We also do not currently implement MACs for communication among inner ring servers. Adding this functionality would also change our results slightly, although the computation already performed to compute threshold signatures is so large in comparison that we believe the change will be small.  

Lastly, our current signature scheme is a non-proactive threshold version of RSA developed by Shoup [Sho00]. There exist proactive schemes that have similar performance. We implemented Shoup’s algorithm in Java, Java with C, and Java with C and assembly language. C and assembly were used through the Java Native Interface (JNI) mechanism. Our Java with C implementation uses libcrypto3, while our Java with C and assembly language implementation uses libgmp4. The latter library contains hand-tuned assembly code for many numerical operations, notably modular exponentiation, a common operation in RSA-based schemes. Table 1 shows the average time to generate a single share of a 512 bit signature using each of our implementations.

Table 2: Basic system parameters of the nodes used in our nationwide tests.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processor(s)</th>
<th>Frequency</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rice</td>
<td>AMD Athlon</td>
<td>800 MHz</td>
<td>256 MB</td>
</tr>
<tr>
<td>UW</td>
<td>Dual Pentium III</td>
<td>930 MHz</td>
<td>1 GB</td>
</tr>
<tr>
<td>GA Tech</td>
<td>Dual Pentium III</td>
<td>800 MHz</td>
<td>512 MB</td>
</tr>
<tr>
<td>UTA</td>
<td>Dual Pentium IIs</td>
<td>300 MHz</td>
<td>256 MB</td>
</tr>
</tbody>
</table>

Table 5: Pairwise ping times between the clusters.

6 Results

In this section, we present a detailed performance analysis of several of the main subsystems of the OceanStore prototype. We begin with a detailed analysis of the write performance of Pond.

6.1 Write Performance

Our write performance numbers were gathered in two experiments. In the first, a single client submits writes of various sizes to a four-node inner ring, one at a time, and measures the time from before the request is signed until the signature over the result is checked. We perform 1,000 writes to warm the JVM, pause for ten seconds,
Figure 5: The mean and standard deviation of thirty-one, 64 byte ping times in milliseconds between the nodes used in our nationwide tests.

<table>
<thead>
<tr>
<th>Source</th>
<th>U. TX</th>
<th>GA Tech</th>
<th>Rice</th>
<th>UW</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCB</td>
<td>45.3 (0.75)</td>
<td>56.5 (0.14)</td>
<td>49.6 (3.1)</td>
<td>20.0 (0.11)</td>
</tr>
<tr>
<td>UTA</td>
<td>-</td>
<td>24.1 (0.49)</td>
<td>8.45 (1.5)</td>
<td>61.7 (0.22)</td>
</tr>
<tr>
<td>GA Tech</td>
<td>-</td>
<td>-</td>
<td>27.7 (2.2)</td>
<td>59.0 (0.20)</td>
</tr>
<tr>
<td>Rice</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>61.5 (0.69)</td>
</tr>
</tbody>
</table>

Figure 6: *Update latency vs. update size.* The latency of a single update is roughly linear in its size; the slope is a factor of hashing and fragment generation speed. The stair-step nature of the lines is caused by the fact that updates are performed on a block granularity.

Figure 7: *Delaying the archive.* Delaying the archive improves median latency, but increases the variance significantly. See text for details.

and then perform 1,000 more, with 5 ms between the response from one write and the request for the next. Figure 6 shows the median latencies; the error bars show the 0th and 95th percentiles.

The lowest line in Figure 6 shows the latency of the archival subsystem alone. The subsequent two lines show the latency for a full update when the archive is not run, and when it is run before returning the result to the client (called “inline” mode), using 512 bit public keys. The upper two lines show the same data using 1024 bit keys. While 512 bit RSA keys do not provide sufficient security today, we show this data to illustrate the effect of increased processing power on our results. A 1024 bit key operation takes approximately four times as long to perform as a 512 bit one, so the change from 1024 bit keys to 512 bit ones represents an expected speedup due to two cycles of Moore’s law (roughly 36 months).

Figure 7 shows the latency in the inline archive case from Figure 6 together with the latency when the archive is run, but the response is returned to the client before the archive has completed (called “delayed” mode). This latter technique reduces client perceived latency, but increases its variance. A separate Byzantine agreement is used to certify the results of the archival process for each update, and the client request may arrive in the middle of this agreement, causing it to be delayed. The stair steps in the curve show the number of items waiting to be agreed upon when the request arrives. Batching the archival result agreement with one over a subsequent request would presumably remove this disparity; we plan to investigate this optimization in the future.

Our second write performance experiment measures the maximum sustainable write throughput of a four-node inner ring. In this experiment, forty clients perform synchronous updates of various sizes against distinct data objects all hosted by the same inner ring. The clients create their objects, synchronize themselves and then begin a 100 second period in which they all write simultaneously. We measure the number of completed updates during this period and multiply by the update size to compute the total bandwidth. Using more clients failed to increase the throughput.

Figure 8.a shows the operations per second as a function of the update size and archival policy, while Figure 8.b shows the resulting throughput in kilobytes per second. For all sizes the operations per second is bounded above by the speed of the threshold signature generation, so for small updates our throughput is quite low. As the clients perform larger update sizes, however, through-
put increases rapidly, and signature generation becomes less important, as can be seen by the converging 512 bit and 1024 bit lines. Implementing batching, where many requests are agreed upon with a single Byzantine agreement, should reduce the minimum update size needed to achieve high throughput. Castro and Liskov observed a similar result [CL00]; we plan to implement this feature in the future.

In the archive case, the throughput of the inner ring on large updates is limited by the throughput of the archival component, though believe we could further tune our archive implementation to correct this problem. Since archiving yields far higher durability than simple replication for an equivalent amount of storage, it is desirable to be able to run the archive at all times. We thus plan to focus on this tuning in our future work.

6.2 Read Performance

Our archival read performance numbers were from an experiment that read data solely from the archive. The experiment has two phases. In the first, we populate the archive by submitting write events of various sizes to a four-node inner ring. In the second phase, a single client submits read events synchronously, and measures 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. Figure 9 shows the median latencies; the error bars show the 0th and 95th percentiles. The experiments used a rate $r = \frac{m}{n} = \frac{16}{32}$ code, i.e. $m = 16$ fragments were required to reconstruct a block. When requesting a block, we requested an additional four fragments to reduce the latency and variance of response times. Latency grows linearly with read size, and response time is slightly longer than the time to the 16th slowest fragment response.

6.3 Tapestry

We now analyze the performance of the Tapestry routing layer. The aggregate behavior of a Tapestry network has been analyzed elsewhere [ZJK01]; here we are primarily concerned with the computational overhead per node in a Tapestry path.

To compute this overhead, we devised a microbenchmark which attempts to isolate the processing overhead for messages of varying sizes. To get accurate measurements without synchronized clocks, we construct a point-to-point, two-node topology and flood the sender’s outbound message queue with messages, pre-
ceded by a marker packet and count. By measuring the arrival time between the marker packet and the last one, we determine the total processing time by the sender and the receiver. To minimize the variance in packet delivery time, we run this experiment on an unloaded network.

Figure 10 shows the results of this experiment. Part (a) shows the bandwidth sustained by Tapestry. Since these experiments were run on gigabit hardware, the bandwidth number is limited primarily by the software implementation. Small messages require proportionally more computation to send, so bandwidth is limited until message sizes reach 16 kB, where it levels off. Because each message is sent individually, there is no overlap in transmission, and we can calculate processing time per message as the inverse of throughput; the result is shown in part (b). Processing time scales linearly with message size, which we attribute to the time required to copy large byte arrays into the message buffer. An empty message requires approximately 56 microseconds to process.

6.4 Dissemination Tree

Our next experiment measures the efficiency of the dissemination tree. Our goal was to show that even a simple method of creating the trees could propagate updates quickly, conserve network bandwidth, and reduce inner ring load. In this experiment, a single writer submits updates to a data object synchronously with no think time, while a variable number of readers continually poll their local secondary replica for the latest version of the object. When they notice a new version has been created, they perform a read.

We ran the benchmark with four inner ring servers and one hundred other OceanStore nodes. We varied the number of nodes sharing the object and the policy used to form the dissemination tree. Under the “no multicast tree” policy, all nodes sharing the object connected directly to an inner ring server to receive update results. Using the “multicast tree” policy, readers ask the system to create replicas in the middle of the network and then connect to those replicas, locating them through Tapestry. Ideally, many replicas will connect to common nodes in the network. The result would be a reduction in the network contention and inner ring load.

We ran the benchmark on an artificial transit-stub network [ZCB96] of 496 nodes. The network had inter-domain latencies of approximately 150 ms and local-area latencies of 10–50 ms. The inner ring was placed on well-connected nodes in different domains in the interior of the network. We then distributed one hundred other nodes randomly throughout the network. The number of replicas sharing the data object was varied from 10–70. Each test runs for 60 seconds; we report the mean and standard deviation of seven tests.

We wanted to show that building a multicast tree conserves network resources. Figure 11 shows the average number of Tapestry hops crossed by dissemination messages with and without a multicast tree. As the number of replicas increases, the benefit of the multicast tree increases. Because the topology Tapestry overlay network is similar to the underlying network topology, a reduction in the number of Tapestry hops indicates a probable reduction in true network distance. We plan to measure and compare these Tapestry-level measurements to true network distance in our future work.

We also observe the effect of the reduced network usage on the performance of the system. Figure 12 shows the latency of the phases of an update, from the initial
Figure 11: Efficiency of dissemination tree. The second tier multicast tree conserves network bandwidth by sending messages across shorter links. The error bars represent the standard deviation of the data.

Figure 12: The effect of the dissemination tree on system performance. A small reduction in network usage can have a significant impact on perceived performance of the system. The dissemination tree shields the inner ring from load allowing the whole system to respond quicker.

request to the arrival of the response at each replica. A modest reduction in network utilization can have a significant affect on the performance of the system. When employing the “no multicast tree” policy, the inner ring nodes must handle load proportional to the number of replicas sharing the object. When this number is large, the load in serving all of the replicas impacts not only the time to deliver results to the replicas, but also the time it takes to create new versions of data.

When using the more advanced dissemination tree, the inner ring nodes can create new versions more quickly, and replicas receive notification of new nodes more quickly on average. We believe that the jitter in the graph is caused when the simple tree creation heuristic results in hot spots (similar to those formed at the inner ring using the “no tree” policy) in the network. Perhaps using more complex techniques and heuristics for creating the dissemination tree [CKK02] would smooth the performance of the system even further.

6.5 Macrobenchmark

To quantify the performance of our prototype on a standard benchmark, we implemented an application which maps the UNIX file system interface onto OceanStore and ran the Andrew benchmark [HKM+88] on it. This section first describes the implementation of the mapping application, then presents the performance results.

The User-Level NFS Server: To provide a UNIX file system interface on top of OceanStore, we implemented a user-level network file system (ULNFS) server as a client application. The Linux kernel allows an NFS server to be run in user space by translating applications calls to NFS ones. For example, an `fwrite` call may be translated to the NFS call `WRITE`, which the kernel then forwards to the user-level server. Our server is implemented in Java, and listens for these kernel messages on an asynchronous UDP socket. It translates the NFS calls to OceanStore client interface calls, which are events passed to the client interface code within the same JVM.

To map the NFS interface to OceanStore, we store files and directories as OceanStore objects. We use the files’ AGUIDs as their file handles. 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 coming-led, 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 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.

Performance Results: To provide comparisons against the NFS based interface to OceanStore, we also ran the user-level NFS daemon packaged with Debian 3.0 and another disk-based user-level NFS daemon we wrote in Java. The Debian daemon provides a comparison with a stable, production NFS server while the disk-based Java implementation is intended to isolate irregularities due to using Java.

The Andrew benchmark is intended to model the kind of accesses commonly performed by file-system users. It contains five phases: it creates a set of directories, copies some source files into them, stats all the files, reads the all files, and compiles the entire tree.

The performance of our server and the two others is shown in Figure 14. On the left are times for a local network; on the right are times for a nationwide test. In the local network case, the four inner ring servers and the client all run on separate machines on our cluster. In the case of the other daemons, the daemon and client always run on separate machines. In the nationwide case, the client is always run at the University of Texas, while the servers run at the other four nodes. The Berkeley server is used in the two non-OceanStore cases. The values shown are the average of at least three runs, and the standard deviation was below ten percent of the mean for all phases.

The local experiments are a worst-case comparison for OceanStore; it is competing in an environment for which it was not designed, against systems tuned for that environment. The no archive case is roughly six times slower than the disk-based Java implementation for 512 bit keys, and twelve times slower for 1024 bit keys. Encouraging, however, is that the inline archive case is only slightly slower than the no archive case.

The nationwide experiments allow a better demonstrations of OceanStore’s strengths. As can be seen from the graph, while OceanStore is still clearly slower on the write-intensive phases, it is much faster on the read-intensive ones. In Phase 3, it is 2.4 times faster in the 1024 bit inline archive case. Overall, the system is only 2.6 times slower than NFS, despite the fact that the inner ring is distributed throughout the country. Also interesting is the diminishing importance of the key size. While public key cryptography is slow, the time used for it is masked by the magnitude of wide area latencies. It is not yet clear why the inline archive case is faster than the no archive case using 1024 bit keys.

7 Related Work

OceanStore builds on the collective experiences of many individual research efforts. One of the first systems to tackle wide-area distributed storage was AFS [Sat90]. AFS gains scalability by caching files locally and deferring object updates until files were closed. Bayou [DPS+94] and Coda [KS92] use replication to improve availability at the expense of consistency and introduce specialized conflict resolution procedures. Sprite [NWO88] also uses replication and caching to improve availability and performance but has consistency guarantees that incur a performance penalty in the face of multiple writers.

Gray et. al. argue against promiscuous replication in [GHOS96]. 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 secondary replicas are similar to transactional caches; in the taxonomy of [FCL97] our algorithm is detection-based and per-
forms its validity checks at commit time. In contrast to similar systems, our merge predicates should decrease the number of transactions aborted due to out-of-date caches.

Second-generation peer-to-peer systems are constructing reliable, self-organizing storage from large numbers of unreliable components. FarSite [BDET00] aims to build an enterprise-scale distributed file system from untrusted components. Its goals include Byzantine fault-tolerance and high availability. The PAST project [RD01b] is producing a global-scale storage system of read-only data using replication for durability. The cooperative file system (CFS) [DKK+01] also targets wide-area, read-only storage. Both PAST and CFS provide probabilistic guarantees of performance and robustness but are not optimized for write traffic.

Intermemory [GY98] is a large-scale, distributed, fault-tolerant archival system that encrypts and erasure-encodes data. It stores fragments on untrusted machines and locates them via an enhanced-DNS look-up scheme. Various incarnations of MojoNation5 also use erasure codes to secure information.

Freenet [CSWH00] and Free Haven6 are publishing systems in support of free expression focusing on publisher anonymity and object authentication. MojoNation7 distributes proprietary content and consequently focus heavily on security, authenticity of data, and access control. Mnemosyne [HR02] uses Tapestry to provide steganographic storage by spreading erasure-coded blocks of data throughout the infrastructure; since data can be overwritten by other users, information must be refreshed at a regular rate.

Other projects, such as Chord [SMK+01], CAN [RFH+01], and Pastry [RD01a], are developing techniques for decentralized object routing and location. All of these routing layers share the goal of utilizing a distributed directory scheme to locate objects in the wide area.

8 Conclusions and Future Work

We have described the first OceanStore prototype, called Pond. While many important challenges remain, this prototype is a functional subset of the vision discussed in the original OceanStore paper [K+00]. Several features discussed in that work have not yet been implemented; in particular, we have not yet integrated encryption or an SDSI-like [Aba97] PKI into our system. Tentative updates have not yet been implemented, and further research on the automatic construction of the dissemination tree is in order. On the other hand, this work has shown that Byzantine agreement with public key cryptography and continuous archiving are not prohibitively expensive in the wide area. Moreover, the use of a self-organizing multicast tree to push updates to secondary replicas is both possible and seems to greatly reduce load on the primary replicas.

While our prototype is not yet deployed on millions of servers, there appear to be no fundamental impediments to scaling the basic design to significantly larger systems. There is no single point of centralization in the system, and every object has the potential to utilize different (but possibly overlapping) resources for its inner ring, dissemination tree and second-tier caches. Finally, the flexibility inherent in the design should allow automatic algorithms the ability to tune the system for better performance. We look forward to deploying OceanStore on ever larger systems with a more diverse set of applications.

9 Acknowledgements

We would like to acknowledge generous support from IBM which made this work possible. In addition, we want to thank Steve Gribble at UW Seattle, Frans Kaashoek at MIT, Peter Druschel at Rice, Ken McKen- zie at Georgia Tech, and Braxton Thomason at UT Austin for allowing us to run OceanStore experiments on their machines.

References


5See http://www.mojonation.net/.
6See http://www.freehaven.net/.
7See http://www.mojonation.net/.


