Maintenance-Free Global Data Storage

OceanStore, a global storage infrastructure, automatically recovers from server and network failures, incorporates new resources, and adjusts to usage patterns.

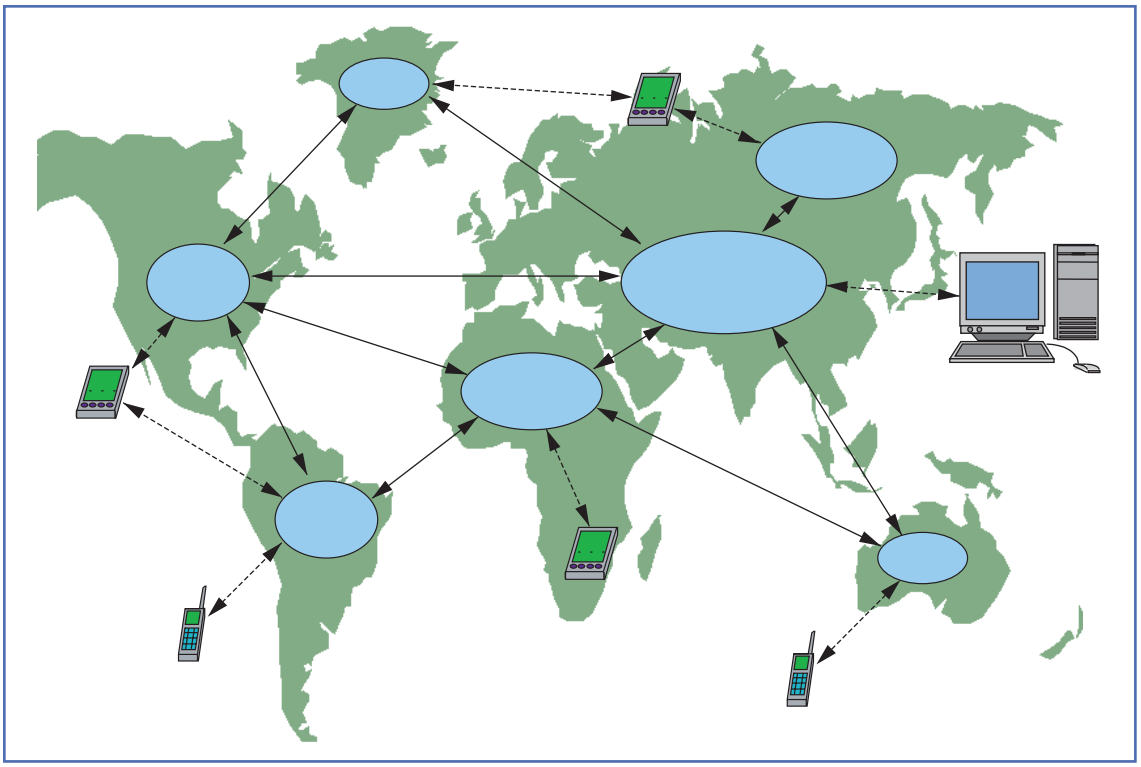
Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz
University of California, Berkeley

The computing world is experiencing a transition from desktop PCs to connected information appliances, which — like the earlier transition from mainframes to PCs — will profoundly change the way information is used. The variety of devices capable of connecting to the Internet is astounding. Personal Data Assistants (PDAs), cellular phones, and even cars have Internet connectivity. Given the proliferation of connected devices, it's natural to address their common need for persistent storage by means of an Internet-based, distributed storage system. Such a system would let devices transparently share data and preserve information even when devices are lost or damaged (for a discussion of applications that will benefit from such a storage infrastructure, see the "Applications for Global Storage Systems" sidebar on p. 44). To handle all of the world's client devices, such a storage system must eventually consist of thousands or millions of Internet-connected servers. Consequently, the greatest obstacle to building and deploying such a system is manageability. Already, single-site data storage systems require carefully orchestrated server configurations, an art understood only by highly paid experts. In a global-scale, distributed storage system, the difficulties increase by many orders of magnitude.

Two distinct types of management are readily identified. *Storage-level management* is the set of operations required to maintain the integrity and access performance of raw data, without reference to the semantics of the information that data represents. Proper storage-level management yields a stable set of bits (or *stable store*) that can be utilized by applications in a variety of ways. *Information-level management,* on the other hand, is the set of operations involved in sorting, classifying, and organizing raw data. Consequently, error correction codes are an aspect of storage-level management, while sophisticated indexing is an aspect of information-level management.

This article explores mechanisms for storage-level management in OceanStore,[1] a global-scale utility infrastructure, designed to scale to billions of users and exabytes of data. A prototype of the OceanStore system is currently under con-

 1089-7801/01/$10.00 ©2001 IEEE



over many servers; only a fraction of the fragments are needed to reconstruct the original block. This coding technique introduces several concerns. One such issue is data integrity: The system must ensure that a reconstructed object is an exact copy of the original, despite failures or malicious corruption of fragments. OceanStore addresses this issue by naming each object and its associated fragments by the result of a secure hash function over the object's contents. This value is the object's *globally unique identifier* (GUID). The security of the hash justifies this terminology — our prototype uses a 160-bit SHA-1 hash for which the probability that two out of 10^{14} different objects hash to the same value is approximately 1 in 10^{20}. Because changing an object's content changes its hash value and name, objects named by GUIDs are read-only. OceanStore's archival subsystem manages these read-only objects, as we discuss later. Because the servers can regenerate and check hashes at any time, fragments are unforgeable and *self-verifying*.

Self-maintenance requires fault tolerance and automatic repair.

Another issue introduced by erasure coding is location: When reconstructing an object, the system must locate enough servers storing fragments for that object. OceanStore addresses this problem with Tapestry, an overlay routing and location layer on top of TCP/IP that maps GUIDs to individual servers. A server can advertise many GUIDs, and any GUID can be advertised by multiple servers. Utilizing Tapestry, OceanStore can easily locate the servers containing fragments for a given GUID.

As specified above, durable elements of OceanStore are read-only. OceanStore overcomes this deficiency through *versioning*,[4] the concept that every update creates a new version of a data object. To support versioning, each OceanStore object has a special name called an active GUID. The system then provides a fault-tolerant mapping from an object's active GUID to the GUID of the most recent read-only version of that object. This mapping is accomplished by the *inner ring*, a group of servers working on behalf of that object. Each object has its own inner ring, although physical servers may participate in many different inner rings. The system builds a mapping from human-readable names to active GUIDs with a hierarchy of OceanStore objects, similar to the SDSI framework.[5] Clients locate servers of an inner ring through Tapestry.

An inner ring's servers are kept consistent through a Byzantine agreement protocol that lets a group of $3f + 1$ servers reach agreement whenever no more than f of them are faulty. Consequently, the mapping from active GUID to most recent read-only GUID tolerates server failure and corruption. The system ensures that no more than f of those servers are faulty via the *responsible party*, a group of servers owned by a user's OSP, that provide a crucial, but low-bandwidth, level of oversight to the system.

The inner ring also verifies a data object's legitimate writers, orders updates, and maintains an update history. This history allows for time travel as in the original Postgres database.[4] *Time travel* facilitates information management by providing a universal undo mechanism. Further, storing old versions of objects allows true referential integrity. These are information-level management properties, however, which are outside the scope of this article.

Finally, because reconstructing an object from archival fragments is computationally intensive, it should be performed only when necessary. OceanStore servers that have already reconstructed objects from the archive can advertise them through Tapestry. These objects are *secondary replicas* of the data, distinct from the *primary replicas* stored on the inner ring. An OceanStore client can retrieve an object either by locating a replica from a nearby server or reconstructing it from the archive.

To make object retrieval less costly and minimize total work, an introspective replica management subsystem automatically creates, replaces, and removes replicas in response to object usage patterns.

Mechanisms for Self-Maintenance

A self-maintaining system requires two fundamental properties: fault tolerance and automatic repair. When a system begins to fail or misbehave, the most important system property is its capability to continue operating correctly, though possibly at reduced performance. Such fault tolerance, however, is not sufficient for self-maintenance, because further failures might render correct operation impossible. Instead, a system must, with high probability, detect and repair a failure before further failures cause a catastrophic error or disrupt service. This section describes self-maintenance in four pieces of the OceanStore architecture.

Routing Infrastructure

OceanStore's Tapestry, a self-organizing routing and object location subsystem, provides object

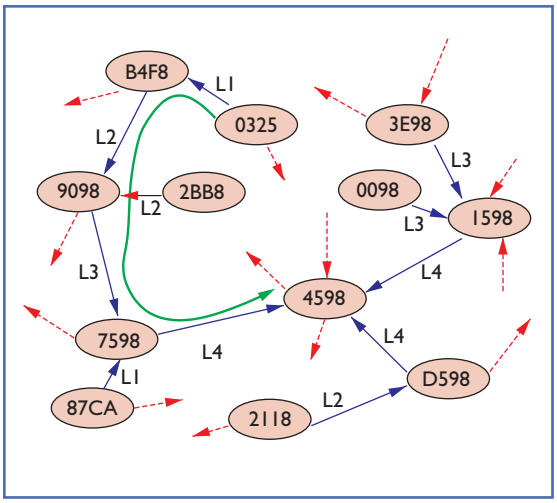


*Figure 2. Tapestry routing example. A potential path for a message originating at node 0325 destined for node 4598. Tapestry routes the message through nodes **** → ***8 → **98 → *598 → 4598, where asterisks represent wildcards. The role each hop plays in the path is marked with a level number.*

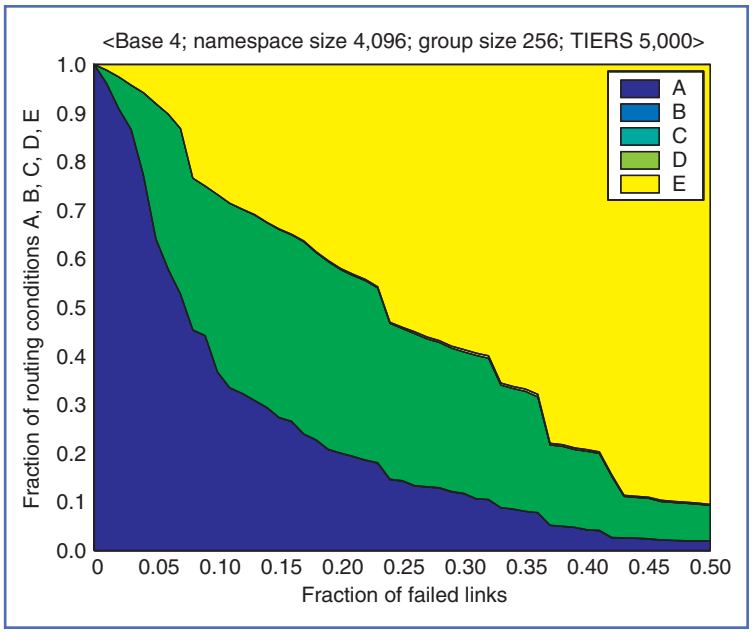


Figure 3. Fault-tolerant routing in Tapestry. This figure shows the benefits of fault-tolerant routing by plotting a reachability measure (fraction of routes available) against the fraction of failed physical links. Regions A through D: physical route exists; Region E: no physical route exists. Region A: both IP and Tapestry successfully route to destination. Region B (barely visible between A and C): IP succeeds, Tapestry fails. Region C: IP fails, Tapestry succeeds. Region D (barely visible between C and E): both protocols fail.

location and node-to-node communication under failure conditions, while transparently handling the automatic insertion and removal of nodes.[6]

Basic routing and location. Tapestry functions as an overlay on top of IP, using a distributed, fault-tolerant data structure to explicitly track the location of all objects. This data structure resembles the hashed-suffix routing structure presented by Plaxton, Rajaraman, and Richa.[7] Each network node can act as a server that stores objects, a client that initiates requests, or a router that forwards messages, or as all of these. Like objects, nodes are assigned unique identifiers called NodeIDs that are location- and semantics-independent.

Tapestry uses local-neighbor maps to incrementally route messages to their destination NodeID, digit by digit. Figure 2 shows an example of such routing. Because the only routing constraint is that each hop match some growing suffix, each routing step in any path can often be satisfied by several destinations. Tapestry keeps pointers to the three closest candidates (in network latency) for each routing entry and, where possible, routes to the neighbor with lowest latency. Nodes early in a path, in particular, generally have many candidates, making early routing hops very fast.

Location through Tapestry works as follows: When an OceanStore server inserts a replica into the system, Tapestry publishes its location by depositing a pointer to the replica's location at each hop between the new replica and the object's *root node*, whose name (NodeID) is a deterministic function of the object's GUID. To locate an object, a client routes a request to the object's root until it encounters a replica pointer, at which point it routes directly to that replica.

Fault tolerance. Tapestry uses its redundant neighbor pointers when it detects a primary route failure while routing a message. Figure 3, which shows the effectiveness of these alternate routing paths, derives from a simulation of a 5,000-node TIERS artificial topology,[8] of which 4,096 nodes participate in the Tapestry protocols. In this simulation, we repeatedly routed packets from randomly chosen nodes in the Tapestry network to 256 destination nodes, while increasing the number of failed links in the physical network. Because IP routing via border gateway protocol (BGP) tables cannot detect failures and find alternate routes in time to redirect packets, we consider packets sent along failed links dropped. Tapestry, on the other hand, uses periodic user datagram protocol (UDP) probes to gauge link conditions, and can immediately deploy backup routes when a failure is detected.

The most striking result in Figure 3 is that Regions B and D are almost invisible, meaning only

Applications for Global Storage Systems

A distributed, fault-tolerant, secure storage infrastructure benefits both very large and small datasets. For example, NASA's Earth Observing System (http://eospso.gsfc.nasa.gov/) is organizing a collection of satellites and data centers to manage scientific data about Earth's climate. The system daily produces more than one terabyte of data, used by more than 10,000 researchers worldwide.

Currently, the data is partitioned across many data centers, each responsible for its own data administration and distribution. A global storage infrastructure would better serve such an unwieldy management scheme, relieving administration and distribution burdens. The infrastructure would ensure that data remain uncorrupted and widely available, and the system would replicate the data, rendering manual backup unnecessary. If part of the infrastructure were compromised or retired, the system would restore replication levels automatically. The system would find requested data regardless of its physical location and create a copy near the user. Researchers could check the data's integrity to ensure that it was not altered during storage or transmission. Furthermore, if users lost their local copy due to administration or hardware failure, the infrastructure could immediately supply a new copy.

At the other end of the spectrum, virtually all PC owners have lost personal data, despite the fact that many own suitable backup devices like tape drives or large removable disks. But such devices must be used on a regular basis and do not protect data from threats such as fire and theft. As the volume and value of data stored on PCs increases, so does the potential loss. This backup problem could be eliminated by a maintenance-free, global-scale storage infrastructure. Data would be replicated and geographically distributed to protect it from hardware failure, configuration errors, and natural catastrophe. Moreover, the data would be stored securely to protect against theft or corruption.

in rare circumstances does Tapestry fail to route to a physically reachable node. As failures increase in frequency, Tapestry greatly increases the probability of successful routing. Even when half the physical links are broken, Tapestry has a 10 percent chance of reaching any destination node.

Besides storing multiple destinations per hop, Tapestry deterministically chooses multiple independent root nodes for each object to provide additional redundancy in the location protocol. The result trades off reliability and bandwidth, because location queries are sent to each root in parallel. In a network with s roots, and assuming a large-scale network partition that divides the network fairly evenly, it is likely ($P \approx 1 - (1/2)^s$) that the data is available via one of the roots.

Automatic repair. Tapestry provides distributed algorithms to adapt to node insertions and removals, fluctuating network conditions and changing user access patterns. To join an existing network, a node chooses a random NodeID by which to identify itself. It then chooses a Tapestry node close in network distance to itself. These nearby Tapestry nodes can be specified by a system administrator or publicized by an external index such as a lightweight, directory access protocol (LDAP) server. Routing to the newly chosen NodeID through this existing node lets the new node find other existing nodes that share incrementally longer suffixes. By copying and optimizing their routing tables at each level, the new node can generate a full routing table. The new node then notifies nearby nodes of its existence so that they can consider it as a more optimal neighbor.

To handle exiting nodes, Tapestry includes two separate mechanisms. A node can simply disappear from the network, in which case its neighbors detect its absence and update routing tables accordingly. Alternatively, a node can use backpointers to inform nodes that rely on it for routing to redirect pointers and to notify object servers for which it stores location information.

The above mechanisms let us dynamically add and remove nodes without manual configuration. Similarly, background processes monitor network conditions to detect suboptimal routes and update routing maps where appropriate.[6]

M-of-*N* Encoding

Erasure codes and self-verifying fragments are the cornerstones of the archival layer's fault tolerance and automatic repair. An erasure code treats input data as a series of m fragments, which it transforms into n fragments, where $n > m$. The resulting code's essential property is that any m of the coded fragments are sufficient to reconstruct the original data. The *rate* of encoding is $r = m/n$. The storage overhead is $1/r$. The system can adjust the durability of information by selecting the rate (and hence overhead).

Fault tolerance. OceanStore divides objects into blocks, producing an erasure-encoded form of each block. An object's inner ring encodes blocks during the update process, sending encoded fragments for each new block to a set of independent storage servers. The inner ring chooses storage servers in a way that minimizes the impact of correlated failures, such as can occur if the storage servers have similar geographic locations or administrative domains. To accomplish this, the inner ring builds models of the independence of storage servers based on historical measurement and information exchanged with other inner rings. Further, correlated failures pose less of a difficulty

to an archival system using a lower rate of encoding (higher overhead). For example, it is extremely unlikely that 24 of 32 randomly placed fragments would all be placed on the West Coast.

Erasure codes allow several servers storing fragments for a particular object to be unreachable at any given time without making the object itself unavailable. With good fragment placement, an encoded block becomes unavailable only in the event of a network partition near the user or a complete partition of the Internet. The archival layer's high tolerance for fragment failures must be supplemented, however, by a good repair mechanism.

Automatic repair. Once fragments have been stored, they must be maintained despite failure. OceanStore offers four maintenance techniques: predicting disk failure, local server sweeps, distributed detection and repair, and global sweep. The first two techniques increase local storage durability; the latter two ensure archive durability.

First, since disks begin to show signs of impending failure before they actually fail,[9] a server can improve the durability of local information by copying it to a new disk before it is lost. Second, a server can periodically verify the fragments it stores. Each fragment contains sufficient additional information such that its contents can by verified by re-hashing the data and comparing the result to the fragment's name. If an error is detected, the server can request the failed fragment's block from the archive, reconstruct the block, and fragment it to recreate the lost fragment.

Third, OceanStore replaces lost information with a distributed detection-and-repair scheme, which is useful because we cannot fully trust servers to maintain the integrity of their own data. A given server could crash, losing all fragments stored on its disk. Or, a server could maliciously delete data. Tapestry contains the location of every fragment; further, this information is self-repairing. Consequently, it's possible to notice — such as at Tapestry root nodes — when the redundancy level for a given object has dropped below a critical level and trigger the recreation and redissemination of lost fragments.

Finally, as a last line of defense, the responsible party periodically sweeps through data under its control, attempting to reconstruct each block. It then regenerates and redisseminates the block's fragments. This process restores lost redundancy to the archive and compensates for malicious or faulty servers.

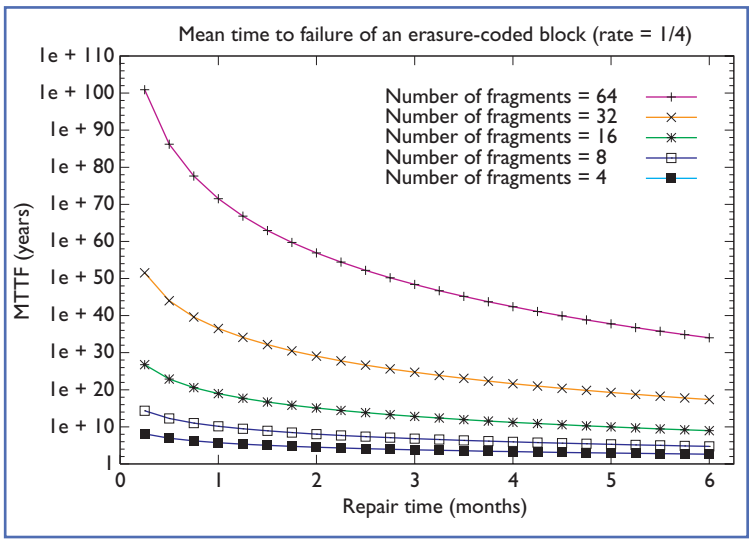


Figure 4. MTTF for a block encoded with a rate-1/4 erasure code. Increasing the number of fragments drastically increases a block's durability while keeping the total storage cost constant. The four-fragment case is equivalent to simple replication on four servers.

Extreme durability. An effective detection and repair scheme can yield data durability unmatched by conventional archiving techniques. A simplified model of OceanStore illustrates the benefits of fragmentation and repair. First, we assume that the archive is a collection of independently failing disks and that failed disks are immediately replaced by new, blank ones. During dissemination, each archival fragment for a given block is placed on a unique, randomly selected disk. Finally, a global sweep-and-repair process scans the system, attempting to restore redundancy by reconstructing each block and redistributing its fragments over a new set of disks. The time period between sweeps of the same block is an *epoch*.

Figure 4 shows a block's resulting mean time to failure (MTTF) for various epoch lengths and numbers of fragments. The probability that a fragment will survive until the next epoch was computed from the distribution of disk lifetimes,[9] supplemented by the assumption that five-year-old disks are at the end of their lifetime, using a method similar to computing the residual average lifetime of a randomly selected disk. The probability that a block can be reconstructed is the sum of all cases in which at least m fragments survive. The MTTF is the mean of the geometric distribution produced from the block survival probability.

The MTTF axis in Figure 4 is logarithmic, indicating that the MTTF of objects scales superlinearly with the inverse of the repair epoch. Although perhaps less clear from the graph, the

Related Work in Distributed Storage

Intermemory[1] 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 lookup scheme. FarSite[2] is an organization-scale distributed file system built from untrusted components. Its goals include Byzantine fault tolerance and high availability. PAST[3] intends to produce a global-scale storage system of read-only data using replication for durability. It includes Pastry, a location component that shares many of Tapestry's properties, described in the main text. CAN[4] and Chord[5] are also examining wide-area location systems.

Other systems have more limited goals. Free Haven (http://www.freehaven.net/) and Freenet (http://freenet.sourceforge.net/) are publishing systems to support free expression; they focus on publisher anonymity and object authentication. Systems such as i-drive (http://www.idrive.com/) and Xdrive (http://www.xdrive.com/) provide available Web-based storage and file-sharing solutions. Finally, systems like Scale Eight (http://www.s8.com/) and Mojo Nation (http://www.mojonation.net/) distribute proprietary content and consequently focus heavily on security, authenticity of data, and access control. Mojo Nation uses Swarm, a peer-to-peer distribution model, while Scale Eight relies on a proprietary global infrastructure for distribution.

REFERENCES

1. A. Goldberg and P. Yianilos, "Towards an Archival Intermemory," *Proc. IEEE Int'l Conf. Advances in Digital Libraries* (ADL 98), IEEE Computer Soc. Press, Los Alamitos, Calif., 1998, pp. 147-156.
2. W. Bolosky et al., "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs," *Proc. Sigmetrics*, ACM Press, New York, 2000.
3. P. Druschel and A. Rowstron, "PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility," *Proc. HotOS Conf.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 2001.
4. S. Ratnasamy et al., "A Scalable Content-Addressable Network," *Proc. SIGCOMM Conf.*, ACM Press, New York, 2001.
5. I. Stoica et al., "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *Proc. SIGCOMM Conf.*, ACM Press, New York, 2001.

MTTF also scales exponentially with the number of fragments. In our simple failure model, given 64 fragments and a repair time of six months, an object has an MTTF of more than 10^{35} years. Simple replication, on the other hand, produces an MTTF of less than 35 years with the same storage cost and repair epoch.

This model analyzes only the sweep-and-repair process and does not include the other mechanisms, specifically the distributed detection and repair mechanism. These more-efficient repair techniques greatly increase the archive's durability and, further, reduce the required frequency of global sweep and repair.

Byzantine Update Commitment

An object's inner ring is responsible for generating new versions of the object from client updates and consistently mapping the object's active GUID to the GUID of its latest version. Each inner ring makes its decisions through a Byzantine agreement protocol.

Fault tolerance. Byzantine agreement guarantees fault tolerance even in the presence of malicious replicas; specifically, f faulty machines can be tolerated by an inner ring of size $n = 3f + 1$. However, Byzantine protocols require $O(n^2)$ messages to be exchanged for every modification of data and were previously believed to be too inefficient for practical use. This is no longer so for several reasons.

First, recent improvements in Byzantine protocols have greatly reduced both the number and size of messages sent during the agreement process. Our protocol is derived from one presented by Castro and Liskov.[10] Their protocol commonly uses only a constant number of phases, rather than a number of phases linear in the size of the ring that characterized earlier protocols. Moreover, for a reasonably large write — say, exceeding four kilobytes — the total bandwidth these messages consume is less than that consumed by sending the write to the replicas. As a result, these algorithms are within a factor of two of the theoretical lower bound for bandwidth use in a replicated system.[1] In fact, Castro and Liskov showed that their protocol was practical for file system use by favorably comparing an implementation of their protocol to a commercial network file system (NFS) server.

Second, the inner ring consists of a few replicas, typically $n < 10$. Because secondary replicas are merely data caches, they don't participate in the Byzantine protocol, but receive consistency information through an application-level multicast tree rooted at the inner ring. Their number does not contribute to the Byzantine protocol's communication costs. This two-tiered architecture is similar to that first presented by Gray and colleagues.[11]

Castro and Liskov's system did not include provisions for caching data, so to ensure that cached data is valid, we extend their system as follows: Replicas in the inner ring use a threshold signature scheme to collectively sign the results of their decisions. Such a scheme allows a signature to be produced as long as $f + 1$ parties agree to cooperate, while preventing any smaller group from generating a signature. Incorporating such a scheme allows for any party in our system to verify that a data object's content was valid at some time, since by assumption no more than f primary replicas are faulty. Moreover, the inner ring members time stamp each decision, allowing any party to verify

that a data object is both valid and current. Parties can verify that an object is current only as precisely as the system clocks are synchronized. Alternatively, they can determine an object's freshness with certainty by requesting the current sequence number directly from the inner ring.

Automatic repair. Automatic repair of the inner ring means maintaining the Byzantine assumption by ensuring that less than a third of the servers in a ring are compromised or faulty at any time.

A *proactive threshold signature*[12] allows the parties who participate in signing objects to be changed over time without changing the key used to verify a signature. This technique allows the responsible party (mentioned earlier) to replace faulty or corrupted replicas without changing the inner ring's public key, and only this key is necessary to verify an object's data. Because the rest of OceanStore communicates with the inner ring through Tapestry, changing the ring's composition like this will not affect other servers' ability to communicate with it. This information-hiding makes it possible to change the set of servers constituting the inner ring without affecting the rest of the system. Because there could be many secondary replicas for a given object, this property is essential for efficient automatic repair.

We maintain the Byzantine assumption by continuously altering the set of servers that participate in an inner ring. We assume that the fraction of faulty or corrupted servers in the world is much less than one third. Given this assumption, replacing any server in the inner ring with a random server from the pool of all servers will likely reduce the number of faulty servers in the inner ring. This replacement can be initiated by the responsible party discussed earlier, and so long as such replacements occur slightly faster than the average rate at which servers within the ring fail, the Byzantine assumption remains valid. Further model extensions, the subject of ongoing research, let the inner ring safely manage its own membership with minimal outside assistance.

Introspective Replica Management

OceanStore allows data caching at any network location. Clearly, the service time of user requests depends on the network distance to the nearest replica, as well as its load. More subtly, replica locality affects service availability when network connectivity is intermittent.

The cost and complexity of manually administering 10^{14} objects to ensure locality and acceptable levels of replication is prohibitive. Instead, we delegate this task to the introspective replica management subsystem.

Introspection is an architectural paradigm that mimics biological adaptation. The introspection layer in OceanStore contains the tools for event monitoring, event analysis, and self-adaptation. It also provides tools for automatic collaboration among introspective modules on different servers.

Introspective modules on servers observe network traffic, measuring access traffic generated by nodes within a few hops. When a server detects heavy demand for an object, it creates a new local replica to handle those accesses. Alternately, clients can request replicas on nearby servers if their perceived quality of service drops below acceptable levels. Additionally, Tapestry root nodes can detect requests that travel a great distance and suggest locations for new replicas. When a replica falls into disuse, a server can remove it from the network and reclaim its resources.

OceanStore allows data caching at any network location.

If a replica suddenly becomes unavailable, the nearby clients will continue to receive service from a more distant replica, due to the fault-tolerant properties of the routing and location mechanism. This failover produces additional load on distant replicas, which the introspective mechanisms detect. OceanStore then creates new replicas to stabilize the system. These activities satisfy the requirements for both fault tolerance and automatic repair.

Bringing It All Together

Against the backdrop of these mechanisms, we can trace the life of a typical server from its integration into OceanStore to its eventual removal.

Integration. To place itself in OceanStore, a new server needs only the address of at least one known Tapestry node; it then weaves itself into the routing and location layer. Next, the server begins advertising its services and the roles it can assume to the world. For example, a server with extra processing cycles and spare storage could advertise its willingness to become an inner ring member or to store new archival fragments. These roles are either specified by administrative parameters or follow recommendations made by its introspective component. For example, a node is unlikely to accept new archival fragments when its available

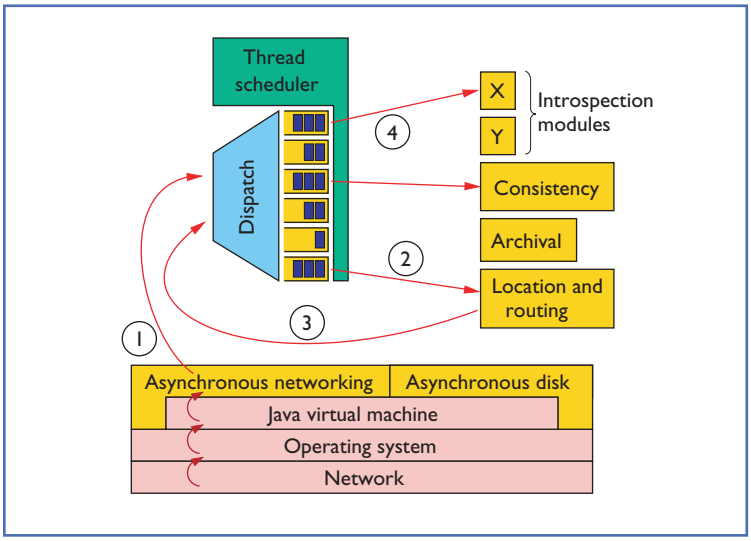


Figure 5. OceanStore server architecture. Numbered arrows show the sample path of one message through the system: (1) Message passes from the network through the OS and JVM to the asynchronous I/O layer; (2) Routing module verifies that message has reached its final destination; (3) Same module then extracts and dispatches message contents; (4) Consistency module receives and processes event; meanwhile, event is sent to an introspective module.

storage falls under acceptable limits. Additionally, for a node to perform certain roles, it must establish a level of trust with the remainder of the system. For example, to become a member of an inner ring, a node must be recommended by some user's OceanStore service provider. This recommendation would be based on the belief that the given node would fail independently of the other members of the given inner ring.

Removal. A server can be removed from OceanStore if it becomes obsolete, needs scheduled maintenance, or experiences component failures. When possible, the server runs a shutdown script to inform the system of its imminent removal. Even in the absence of this announcement, OceanStore will detect and correct for the server's absence. Tapestry nodes that depend on the server for routing will promote secondary routers and find new backups. Servers in primary rings will elect replacements from the global pool of available servers. The roots for archived objects for which the server was storing fragments will cause server pointers to time out, notifying the objects' responsible parties, and introspective mechanisms will ensure regeneration and redissemination of fragments if redundancy falls under an acceptable level.

Prototype Architecture

Figure 5 depicts the event-driven server architecture of the OceanStore prototype under construction at UC Berkeley. This prototype is layered on SEDA (http://www.cs.berkeley.edu/~mdw/proj/seda/) and implemented in Java. Although a complete system is not yet operational, many components are already functioning in isolation. The Tapestry routing infrastructure is in small-scale test, including a multicast facility. A prototype NFS communicates with the Byzantine inner ring and can run the Andrew file system benchmark. Further, a stand-alone version of the archival system is operational as a backup system. We are presently integrating these components. Several clients are currently under construction, including an NTFS file system, a full-function backup management facility, and an HTML gateway.

Conclusion

OceanStore provides a global-scale, distributed storage platform through adaptation, fault tolerance, and repair. The only role of human administrators in our system is to physically attach or remove server hardware. Of course, an open question is how to scale a research prototype in such a way to demonstrate the basic thesis of this article — that OceanStore is self-maintaining. In the coming months, we will demonstrate this premise in a number of ways, including large-scale simulations with fault injection and wide-area deployment of functional OceanStore server components. The allure of connecting millions or billions of components together is a hope that aggregate systems can provide scalability and predictable behavior under a wide variety of failures. The OceanStore architecture is a step toward this goal.

References

1. J. Kubiatowicz et al., "OceanStore: An Architecture for Global-Scale Persistent Storage," *Proc. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, ACM Press, New York, 2000.
2. W. Bolosky et al., "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs," *Proc. Sigmetrics*, ACM Press, New York, 2000.
3. J. Bloemer et al., "An XOR-Based Erasure-Resilient Coding Scheme," tech. report TR-95-048, The Int'l Computer Science Inst., Berkeley, Calif., 1995.
4. M. Stonebraker, "The Design of the Postgres Storage System," *Proc. 13th Int'l Conf. Very Large Databases*, Morgan Kaufmann, San Francisco, 1987.
5. R. Rivest and B. Lampson, "SDSI — A Simple Distributed Security Infrastructure," manuscript presented at CRYPTO 96 Rump session, 1996; details available at http://theory.lcs.mit.edu/~cis/sdsi.html.

6. B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," tech. report UCB/CSD-01-1141, Apr. 2001, Univ. of California at Berkeley.
7. C. Plaxton, R. Rajaraman, and A. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," *Proc. ACM Symp. Parallel Algorithms and Architectures,* (SPAA 97), ACM Press, New York, 1997.
8. M. Doar, "A Better Model for Generating Test Networks," *Proc. IEEE Globecom*, IEEE Press, Piscataway, N.J., 1996.
9. D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach,* 3rd ed., to be published Mar. 2002, Morgan Kaufmann, San Francisco.
10. M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. Usenix Symp. Operating Systems Design and Implementation* (OSDI 99), Usenix Assoc., Berkeley, Calif., 1999.
11. J. Gray et al., "The Dangers of Replication and a Solution," *Proc. ACM SIGMOD Conf.*, ACM Press, New York, 1996.
12. B. Barak et al., "The Proactive Security Toolkit and Applications," *Proc. ACM Conf. Comm. and Computer Security*, ACM Press, New York, Nov. 1999.

Sean Rhea is a PhD student at the University of California, Berkeley, and is the principal architect of OceanStore's inner ring design. His research interests include wide-area location and routing algorithms and the use of Byzantine agreement protocols in fault-tolerant file systems. He received a BS in computer engineering from the University of Texas, Austin.

Chris Wells, although currently unaffiliated, is one of the principal architects of the archival generation and repair mechanisms of OceanStore. He received a BS in computer science from the University of Kentucky and an MS from the University of California, Berkeley.

Patrick Eaton is a PhD student at the University of California, Berkeley, where he is responsible for OceanStore's second-tier object servers design. His research interests include mechanisms for global-scale data coherence and APIs for multiversioned objects. He received a BS in computer engineering from the University of Illinois, Urbana-Champaign.

Dennis Geels is a PhD student at the University of California, Berkeley, where he is developing efficient learning models for distributed introspection in OceanStore. His research interests include operating systems, computer architecture, and statistical learning theory. He received two BA degrees from Rice University, one in computer science and one in mathematics.

Ben Zhao is a PhD student at the University of California, Berkeley, where he is the principal architect of the Tapestry network infrastructure. His interests include global-scale, fault-tolerant, and location-independent routing, as well as service discovery services and in-memory XML databases. He received a BS in computer science from Yale University and an MS from the University of California, Berkeley.

Hakim Weatherspoon is a PhD student at the University of California, Berkeley, where he is one of the principal architects of OceanStore's archival generation and dissemination mechanisms. His research interests include global-scale, fault-tolerant archival storage and maintenance. He received a BS in computer engineering from the University of Washington.

John D. Kubiatowicz is an assistant professor at the University of California, Berkeley, where he is the technical lead and principal investigator for OceanStore. In addition to global-scale storage systems, his interests include computer architecture, multiprocessor design, quantum computing, and continuous dynamic optimization. He received two BS degrees from the Massachusetts Institute of Technology, one in electrical engineering and one in physics. He also received an MS and a PhD from MIT in electrical engineering and computer science.

Additional information about OceanStore, including author contact information, is available at http://oceanstore.cs.berkeley.edu/.

Nine good reasons why close to 100,000 computing professionals join the IEEE Computer Society

Transactions on

- *Computers*
- *Knowledge and Data Engineering*
- *Multimedia*
- *Networking*
- *Parallel and Distributed Systems*
- *Pattern Analysis and Machine Intelligence*
- *Software Engineering*
- *Very Large Scale Integration Systems*
- *Visualization and Computer Graphics*

IEEE COMPUTER SOCIETY

computer.org/publications/