HYDRAstor: a Scalable Secondary Storage

Cezary Dubnicki[§], Leszek Gryz[§], Lukasz Heldt[§], Michal Kaczmarczyk[§], Wojciech Kilian[§], Przemyslaw Strzelczak[§], Jerzy Szczepkowski[§], Cristian Ungureanu, and Michal Welnicki[§],
NEC Laboratories America
[§]formerly at NEC Laboratories America, now at 9LivesData, LLC
{dubnicki, gryz, heldt, kaczmarczyk, wkilian, strzelczak, jsz, welnicki}@9livesdata.com, cristian@nec-labs.com

Abstract

HYDRAstor is a scalable, secondary storage solution aimed at the enterprise market. The system consists of a back-end architectured as a grid of storage nodes built around a distributed hash table; and a front-end consisting of a layer of access nodes which implement a traditional file system interface and can be scaled in number for increased performance.

This paper concentrates on the back-end which is, to our knowledge, the first commercial implementation of a scalable, high-performance content-addressable secondary storage delivering global duplicate elimination, per-block user-selectable failure resiliency, self-maintenance including automatic recovery from failures with data and network overlay rebuilding.

The back-end programming model is based on an abstraction of a sea of variable-sized, content-addressed, immutable, highly-resilient data blocks organized in a DAG (directed acyclic graph). This model is exported with a low-level API allowing clients to implement new access protocols and to add them to the system on-line. The API has been validated with an implementation of the file system interface.

The critical factor for meeting the design targets has been the selection of proper data organization based on redundant chains of data containers. We present this organization in detail and describe how it is used to deliver required data services. Surprisingly, the most complex to deliver turned out to be on-demand data deletion, followed (not surprisingly) by the management of data consistency and integrity.

1 Introduction

The enterprise environment places strenuous demands on the secondary storage systems. With ever increasing amounts of data produced and fixed backup windows, there is a clear need for scaling performance and backup capacity appropriately. Different types of data have varying importance which require different classes of reliability and availability and have specific retention periods. Regulatory requirements (SOX, HIPPA, the Patriot Act, SEC rule 17a-4(t)) demand security, traceability and data auditing. Strict data retention and deletion procedures need to be defined and followed rigorously. Failure to present retained data on demand can result in serious business losses, fines and even criminal prosecution. Last but not least, limited IT budgets increase the importance of providing efficient storage by improving storage utilization for backup and archival applications, and by reducing the data management costs.

Substantial progress has been made to address these enterprise needs, as demonstrated by advanced disk-targeted deduplicating Virtual Tape Libraries [2, 3], disk-based back-end servers [43] and content-addressable archiving solutions [4]. However, the exponential increase in the amount of data stored creates new problems not addressed by these solutions. First of all, unlike primary storage, which is usually networked and under common management (e.g. SANs), secondary storage consists of a large number of highly-specialized dedicated components, each of them being a storage island requiring customized, elaborate, and often manual administration and management. As a result, large fraction of the total cost of ownership (TCO) can still be attributed to management of more and more of secondary storage components [1, 17, 21]. Moreover, fixed capacity assignment to each storage device results in poor capacity utilization. Duplicate elimination in these islands of storage is similarly limited in scope which compounds the inefficiency. Finally, since each of secondary storage devices offers fixed, limited performance, reliability and availability, the high overall requirements of enterprise secondary storage in these dimensions can be met only by implementing complex in-house solutions.

Fortunately, new technology and previous research results provide building blocks for a solution addressing these problems. The content-addressable storage paradigm [4, 24, 43] enables cheap and safe implementation of duplicate elimination. Distributed hash tables [12, 20, 25, 27, 31, 42] allow for building scalable, failure-resistant systems and extending duplicate elimination to a global level. Erasure codes can add resiliency to the stored data with fine-grain control between required resiliency level and resulting storage overhead. Hardware and pricing trends are also critical for enabling HYDRAstor. The capacity of SATA drives and the performance of new multi-core CPUs increase even as the costs of these components fall. Together, these trends provide the building blocks needed for systems like HYDRAstor at a very reasonable cost.

Other work applicable include research on self-management [13, 33], monitoring [35], and on-line reconfiguration [30] and upgrade [7]. Although all of these elements facilitated building HYDRAstor, the task proved to be much more complex than we originally envisioned and required a significant amount of original research. The effort often felt like trying to design and construct a building given just bricks and stones.

HYDRAstor [23] is a commercial secondary storage solution for the enterprise addressing shortcomings discussed earlier. It consists of a back-end architectured as a grid of storage nodes delivering scalable capacity and a front-end consisting of a layer of access nodes scaled for performance. In this paper, we concentrate on the design of the back-end grid which supports capacity sharing between all clients and types of data, for example, back up images or archival data. This sharing together with system-wide deduplication allow for highly efficient use of storage capacity. The system is highly-available, as it supports on-line extensions and upgrades, tolerates multiple disk, node and network failures, rebuilds the data automatically after failures and informs users about recoverability of the deposited data. The reliability and availability of the stored data can be dynamically adjusted by the clients with each write, as the back-end supports multiple data resiliency classes.

This paper makes the following contributions. First, it presents the HYDRAstor as a concrete commercial implementation of scalable secondary storage system addressing today's enterprise needs. Second, it discusses in detail the HYDRAstor data organization and how it is used to implement advanced data services like global duplicate elimination, on-demand deletion, and data integrity management. Third, it contains an evaluation of the HYDRAstor that demonstrates effectiveness of its implementation.

The remainder of this paper is organized as follows. Section 2 describes the system's functionality including the programming interface. Section 3 contains a high-level discussion of the back-end design. It establishes context for the next section, 4, which discusses requirements on data organization and the resulting solution. Section 5 illustrates how this organization is used to deliver data services like data rebuilding and distributed data deletion. Section 6 presents evaluation of the system. Related work is discussed in Section 7, whereas conclusions and future work are given in Section 8.

2 Functionality

The back-end has been designed as a vast data repository, allowing for storing and extracting streams of data with high throughput. Internally, it consists of a potentially large number of independent nodes presented externally as a single system image. The back-end is designed to scale up to thousands of dedicated nodes which could provide hundreds of petabytes of storage. The primary deployment target is the data center.

From the beginning, the HYDRAstor back-end was intended to provide a foundation for a commercial product. Therefore, one of the design targets has been to support not only tailor-made new applications, but also commercial legacy applications, as long as they use streamed data access. To that end, the system does not define one fixed access protocol, instead it is flexible to allow support for legacy applications using standards like file system interface as well as for new applications using highly-specialized access methods. New protocols can be dynamically added to an online system by loading a new protocol driver without disrupting any client using the existing protocols.

One of the primary design goals has been to ensure continuous operation of the system, limiting or eliminating impact of upgrades, extensions and failures. The distributed architecture enhances system availability by allowing online software or hardware upgrades in most cases, eliminating the need for costly downtime. Moreover, the system is capable of automatic self-recovery in case of hardware failures (disk, network, power loss), and even from some of software failures. The system works correctly in the presence of up to a specific configurable number of fail-stop and intermittent hardware failures. The system does not handle Byzantine failures which have a very low probability of actually occurring in a real data center and would add significant overhead. However, the system has several layers of data integrity checking to detect data corruption.

Another important function of the system is to ensure high data reliability, availability and integrity. Each block of data is written with a user-selected resiliency level which allows the user to choose how many concurrent disk failures the block can survive. This is achieved with erasure coding each block into fragments; as shown in [36] erasure codes increase mean time to failure by many orders of magnitude over simple replication for the same amount of space overhead. After a failure, if a block remains readable, the system automatically schedules data rebuilding to bring the resiliency back to the level requested by the user. No permanent data loss remains hidden for long. Global state indicates whether all stored blocks are readable, and if so, how many disk and node failures must happen before data loss occurs.

Secondary storage systems have unique characteristics which influence the design goals. In contrast to primary storage, which often deals with random accesses, these systems are dominated by writes of long data streams. Given the scale of the system, multiple streams will be written concurrently by different clients. Successive streams are often similar to previously written streams which can contain many duplicate blocks. Since all data must be saved during short backup windows, very high write throughput is essential. Read throughput is quite important for restores, but it is not as critical as write throughput in our system since the restores are typically much less frequent and involve reading only a portion of the stored data.

2.1 Programming Model

The back-end programming model is based on an abstraction of a sea of variable-sized, content-addressed, immutable, highly-resilient blocks. A block consists of data and, optionally, an array of block addresses, pointing to previously written blocks. A block's address is derived from the SHA-1 hash of its content (both data and pointers). Blocks are variable-sized to allow for better deduplication; and pointers are exposed to facilitate data deletion implemented as garbage collection. The back-end exports a low-level block interface used to implement new and legacy protocols. This interface allows for a clean separation of the back-end from the front-end which can support a wide range of access protocols.

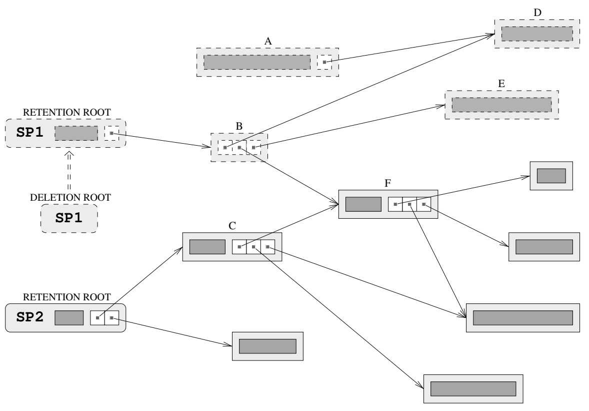

Figure 1: Blocks organized in a directed acyclic graph. Data part of each block is shaded, pointers are not.

Blocks in the back-end form a DAG (directed acyclic graph), as illustrated by Fig. 1. Drivers write trees of blocks, but because of deduplication, these trees overlap at deduplicated blocks and form directed graphs. No cycle is possible in these structures as long as the hash used in the block address is secure. A source vertex in a DAG is usually a block of a special type called *searchable retention root*. In addition to the regular data and the array of addresses, a retention root contains a user-defined *search key* used to locate the block. This key can be arbitrary data. A user retrieves a searchable block by providing its search key instead of a cryptic block content address. For example, multiple snapshots of the same file system can have each root organized as a searchable retention root with search key containing file system name and a counter incremented with each snapshot. Searchable blocks do not have user-visible addresses and cannot be pointed to, so they cannot be used to create cycles in block structures. However, each searchable block has an internal hashkey assigned to it for fast retrieval. Unlike regular blocks, the hashkey of a searchable block is computed only over the search key portion of the block's data.

Fig. 1 shows a set of blocks organized into a DAG with 3 source vertices, 2 of them are retention roots; the 3rd source vertex is a regular block, which indicates that this part of the DAG is still under construction.

The API operations include writing and reading regular blocks, writing searchable retention roots, searching for a retention root based on its search key; and marking a retention root with a specified key to be deleted by writing an associated deletion root, as discussed below. Cutting the data stream into blocks is beyond this interface and is responsibility of the drivers, although we plan to re-evaluate this decision soon.

When writing a block, a driver assigns it to one of a few available *resiliency classes*. Each class represents a different tradeoff between data resiliency and storage overhead: from the low resiliency data class where a block can survive only a single disk failure but has minimum storage overhead, up to the critical data class where each block can be replicated multiple times on different disks and physical nodes. Different resiliency classes are achieved by varying the number of original fragments in the erasure coding scheme (described later).

The system does not provide a way to delete a single block immediately because this block may be referenced by other blocks. Instead, the API allows to mark roots of the DAG(s) which should be deleted. To mark a retention root as dead, a user writes a special block called *searchable deletion root* with the search key identical to this retention root's search key. In Fig. 1, there is a deletion root associated with the retention root SP1. The deletion algorithm marks for deletion all blocks not reachable from the live retention roots, for example in Fig. 1 all blocks with dotted lines will be marked. The block named A will also be deleted because there is no retention root pointing to it, whereas the block named F will be retained, as it is reachable from the retention root SP2 which is still defined as live since it does not have a matching deletion root.

During data deletion, there is a short read-only period, in which the system identifies blocks to be deleted. Actual space reclamation happens in the background during regular read-write operation. Before entering a read-only phase, all blocks to be retained should be pointed by live retention roots.

3 System Architecture

HYDRAstor back-end nodes are built of highly reliable server-grade components. No customized hardware is needed. Detailed description of available hardware configurations is given in Section 6. The number of storage nodes determines the total raw capacity of the system as well as its maximal level of performance. Front-end access nodes can be added to realize this performance up to the limit determined by the current back-end configuration.

Software components of the back-end include the *storage server* and *proxy server*, both implemented as Linux user space processes, and *protocol drivers* implemented as libraries.

Storage servers are organized in an overlay network, with data blocks assigned to each server based on block's hashkey. The details of the overlay are discussed in Section 3.1. Each storage node hosts one or more storage servers. The number of storage servers running on a storage node depends on its resources. The bigger the node, the more servers we run, with each server responsible exclusively for a specific number of this node's disks. Putting multiple servers on one physical node is a simple solution to the problem of harnessing computing power of multicore CPUs.

Proxy servers run on access nodes and export the same block API as the storage servers. A proxy provides services like locating the storage nodes, optimized message routing and caching.

Protocol drivers use the API exported by the back-end to implement access protocols. These drivers can be loaded in the runtime on both storage and proxy servers. Location of a driver depends on available resources and driver resource needs. Usually, resource-hungry drivers like the file system driver are loaded on proxy servers.

3.1 Network Overlay

Since one of our design goals has been scalability, the use of distributed hash tables has been a natural choice. However, because for a distributed storage system both storage utilization and data resiliency are extremely important, we have had additional requirements on a DHT: assurances about storage utilization and ease of integration of the selected overlay network with the data resiliency scheme we have planned to use, i.e. erasure coding. Since none of the existing DHTs allowed for that, we have decided to use a modified version of the *Fixed Prefix Network* (FPN) [12] distributed hash table. FPN makes it possible to maintain very short routing paths for a wide range of the number of nodes and guarantees a minimal level of storage utilization.

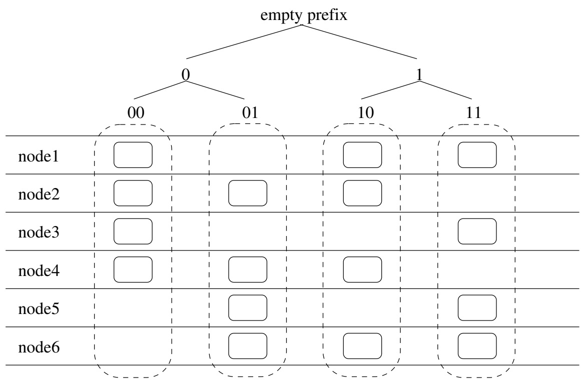

Figure 2: Supernodes and components. 4 supernodes spanned over 6 physical nodes. Each supernode has 4 components, i.e. supernode cardinality is 4.

In FPN, each overlay node is assigned exactly one hashkey prefix used also as an identifier of this virtual node. An FPN node is responsible for hashkeys with prefix equal to this node identifier. All possible hashkeys form a hashkey space. The overlay network strives to keep prefixes disjoint and to cover completely this space, which we call also *prefix space*. The upper part of Fig. 2 shows a prefix tree which has four leaf FPN nodes, dividing the prefix space into four disjoint subspaces.

To meet our DHT requirements, we have extended the original FPN with *supernodes*. A supernode represents one FPN node (and as such, it is identified with a hashkey prefix), but spans several physical nodes to increase resiliency to node failures. Each supernode consists of a fixed number (called *supernode cardinality*) of *supernode components*. Components of the same supernode are called *peers* and are usually placed on separate physical nodes, as show on Fig. 2. Practical supernode cardinality values are in the 4-32 range, and in the commercial HYDRAstor it is set to 12. For a given HYDRAstor incarnation, its supernode cardinality is the same for all supernodes and is constant throughout entire system lifetime.

Supernode peers use a distributed consensus algorithm to decide what change should be applied to the supernode — for example, after node failure, they decide on which physical nodes new incarnations of lost components should be recovered.

3.2 Read and Write Handling

On write, a block of data is routed to one of the peers of the supernode responsible for the hashkey space where the block's hash belongs. For both read and write requests, the peer is deterministically chosen based on the hashkey of the data. Next, this write-handling peer checks if a suitable duplicate is already stored; this process is described in detail in Section 5.2. If a duplicate is found, its address is returned; otherwise the new block is compressed (if requested by a user), fragmented, and its fragments are distributed to remaining peers.

On read, the read handling peer first determines the minimal number of fragments required to reconstruct this block, which is stored in the block's metadata. Next, the read handling peer sends fragment read requests to some of the other peers. If any of these requests times out, all remaining fragments are read. After a sufficient number of fragments have been found, the block is reconstructed, decompressed (if it was compressed), its SHA-1 hash is verified and, in case of successful verification, returned to the user.

In general, sequential reading is very efficient since the blocks are read in the same order that they were written and the individual fragments end up getting prefetched from disk into local memory. Usually, the requested fragment is present in the current component location. However in some cases (for example after intermittent failures), the requested fragment may only be present in one of the previous locations of this component. In such a case, the component directs a distributed search for the missing data. In particular, the trail of previous component locations can be searched in the reverse order.

3.3 Load Balancing

In a distributed storage system like the HYDRAstor back-end, the distribution of data among physical nodes is critical for system survivability, data resiliency and availability, storage utilization, and system performance. For example, placing too many peer components on one machine may have catastrophic consequences if this node is lost. The affected supernode may not recover, because too many components have been lost; and even when it is recoverable, some or even all of the data handled by this supernode may not be readable, due to loss of too many fragments. Also, performance of the system is maximized when components are assigned proportionally to available node resources, since the load on each node is proportional to the prefix space covered by the components assigned to this node.

Our system continuously attempts to balance component distribution over all physical machines to reach a state where failure resiliency, performance and storage utilization are maximized. The quality of a given distribution is measured by a multi-dimensional function prioritizing these objectives, called *system entropy*. Balancing is carried out by each machine, which periodically considers all possible transfers of locally hosted components to neighboring nodes. If the machine finds a transfer that would improve the distribution, it is executed. After a component arrives at a new location, its data is also moved from old location(s) to the new one; but this data transfer happens in the background and may take a long time.

The same entropy-driven balancing is applied to the system when nodes are added or removed from the system.

3.4 Impact of Supernode Cardinality

Selection of supernode cardinality has profound impact on properties of HYDRAstor. First of all, it determines the maximal number of tolerated node failures. The network overlay, but not necessarily user data, survives node failures as long as each supernode remains alive. A supernode survives if at least half of the supernode's peers plus one remain alive so they can reach a consensus.

Supernode cardinality also influences scalability, at least in theory. For a given cardinality, the probability that each supernode survives is fixed; the higher the cardinality the higher the probability of survival. When a system size grows, its number of supernodes also grows, and, as a result, the system reliability decreases, as for the system to be operational we require all supernodes to be alive. However, the practical impact of this limitation is negligible in the target range of system size, because permanent loss of a physical node is very rare, and self-healing reduces the window of vulnerability even when it happens.

Finally, supernode cardinality defines the number of data redundancy classes available. Erasure coding is parametrized with the maximal number of fragments that can be lost while a block remains still reconstructible (standard m-of-n erasure codes with n set to supernode cardinality and m determined by the redundancy class; we use the Cauchy-based Reed-Solomon codes [9]). Since in HYDRAstor the erasure coding always produces supernode cardinality fragments, the tolerated number of lost fragments can vary from one to supernode cardinality minus one (in the latter case we keep supernode cardinality copies of such block). Each such choice of tolerated number of lost fragments defines one data redundancy class. Each class represents different tradeoff between storage overhead (due to erasure coding) and failure resiliency.

4 Data Organization

Proper representation of stored data is critical for meeting reliability, availability and performance targets of HYDRAstor. The system should be able to easily identify the availability of stored data, and in case of a failure, rebuild only the data actually written and only to the requested resiliency level (as opposed to RAID, which rebuilds entire disk even if it contains no valid user data). Since components move between nodes followed by the data transfer, it should be possible to locate and retrieve data from old component locations. When such data is available, it should be transferred instead of being rebuilt, as transfer is a much cheaper operation. Data written in one stream should be placed nearby to maximize write and read performance. Last but not least, the data organization should support on-demand distributed data deletion, in which data blocks not reachable from any live retention root are deleted and the space occupied by them is reclaimed.

4.1 Synchruns and Synchrun Components

As discussed earlier, we use erasure coding for data redundancy. Resulting fragments of one block are distributed to peer components of the supernode responsible for this block. The basic logical unit of data management in HYDRAstor is the *synchrun*, containing a limited number of blocks written consecutively by one write-handling peer component.

A synchrun is analogous to a stripe in a RAID group since both allow faster reads and writes of continuous data faster than any single disk can do. Unlike a RAID stripe, a synchrun is also the basic block that is used for data balancing and load management as described below. Since writing a block really means writing a supernode cardinality of its fragments, each synchrun is represented by supernode cardinality of *synchrun components*, one for each peer. For the i-th peer of a supernode, the corresponding synchrun component contains all i-th fragments of the synchrun blocks. A synchrun is a logical structure only, but synchrun components actually exist on corresponding peers.

4.2 Chains of Containers

At any given time, each write-handling peer writes block fragments to exactly one synchrun. As a result, all such synchruns can be logically ordered in a chain, with the order determined by the write-handling peer. Synchrun components are placed in a data structure called *synchrun component container (SCC)*. Each SCC can contain one or more chain-adjacent synchrun components, and as a result, SCCs form also chains similar to synchrun component chains.

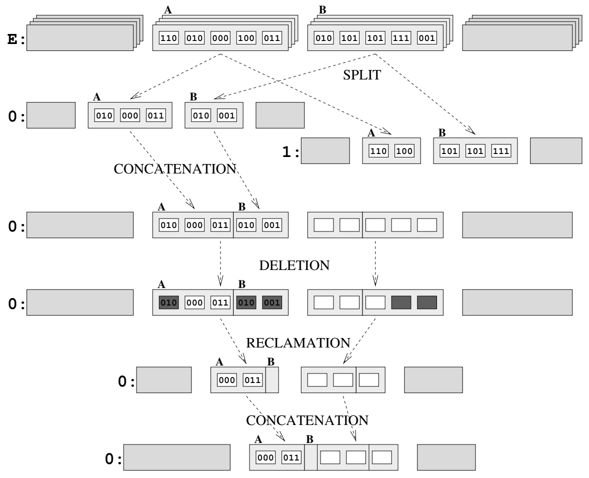

Figure 3: Data organization with synchruns and synchrun containers.

The upper row in Fig. 3 shows synchruns A and B that belong to the empty prefix supernode which covers the entire hashkey space. Each synchrun component is placed here in one SCC, with its individual fragments represented by smaller boxes inside the SCC. SCCs with synchrun components of these synchruns are shown as rectangles placed one behind the other. A chain of synchruns is represented by the supernode cardinality of SCC chains, we call them *peer SCC chains*. In the remainder of the Fig. 3 we show only one such peer SCC chain.

Peer SCC chains are normally identical with regards to the synchrun components' metadata and the number of fragments they hold, but there are occasional differences caused by node failures which cause holes in the chains. This chain organization allows for relatively simple and efficient implementation of required features. For example, if the number of peer chains without any holes is not lower than the number of fragments needed to reconstruct each block, then we infer that the data is available (i.e. all blocks are reconstructible). In such way, determination of data availability can easily be made for each redundancy class.

Each supernode will eventually be split as more data is stored or as more nodes are added to the system. This is a regular FPN split which results in two new supernodes with prefixes extended from their ancestor prefix with, respectively, 0 and 1. After a supernode split, each synchrun in this supernode is also split, with fragments distributed between them based on their hash prefixes. The second row of Fig. 3 shows two such chains, one for the supernode with the prefix 0, and the other with the prefix 1. As a result of the split, fragments from synchruns A and B are distributed to these two chains. The system now has 4 synchruns, each approximately half the size of the orignals.

The system strives to maintain a limited number of local SCCs, and merges adjacent synchrun components into one SCC (as shown on the third row of Fig. 3) until the maximum size of an SCC is reached. By limiting the number of local SCCs, the system can keep their metadata cached in RAM which enables fast determination of actions needed for providing data services. The target size of an SCC is a configuration constant (usually set well below 100 MB), so multiple SCCs can be read into the main memory. These SCC concatenations are loosely synchronized on all peers, so peer chains look the same. A similar operation is needed after deletion, shown in the remaining rows of this figure and discussed later in Section 5.3

This data organization is relatively simple in a static system, but it becomes quite complex due to the dynamic nature of the HYDRAstor back-end. For example, when a peer is transferred to another physical node because of load balancing, its chains are transferred in the background to a new location, one SCC at a time. Similarly, after a supernode split, not all SCCs of the supernode are split immediately; instead we run background operations adjusting chains to the current supernode locations and shape. As a result, in any given moment, we may have chains partially-split, partially present in previous locations of this peer, or both. After failure, we may have serious holes in some of the chains. Fortunately, since peer chains describe the same data, we have supernode cardinality chain redundancy in the system, so usually there is a sufficient number of complete chains. This chain redundancy allows for reasoning about the data in the system even in the presence of transfers/failures. Additionally, more refined algorithms are used in some cases, constructing chain coverage from chain parts present on different peers.

5 Data Services

Based on the data organization described above, HYDRAstor efficiently builds data services like identification of the recoverability of data, deletion and space reclamation, locating data in the network, data deduplication and others. Given a detailed description of all of these features is beyond the scope of this paper, but this section will present a sketch of the data rebuilding, deletion and duplicate elimination services.

5.1 Data Rebuilding

When a node or disk fails, the SCC's residing on that node or disk are lost. As a result, the redundancy of the data blocks with fragments belonging to these SCCs is at best reduced below the level requested by users when writing these blocks. In the worst case, a block may be lost completely if not enough fragments survive. To keep the block redundancy at the desired levels, the system scans SCC chains looking for holes and schedules data rebuilding as background jobs for each missing SCC.

Multiple peer SCCs can be rebuilt in one rebuilding operation. Based on SCC metadata, the minimal number of peer SCCs needed for rebuilding is read by the peer that is in charge of rebuilding. This peer does bulk erasure decoding and encoding to restore the missing fragments. Next, the rebuilt SCCs are sent to the current target locations. Before SCCs are rebuilt, all input SCCs are made to look the same, i.e. required splits and concatenations are performed first. This requirement allows for fast bulk rebuilding as measured in Section 6.

5.2 Duplicate Elimination

Duplicate elimination can be classified in many dimensions: (1) the granularity of the deduplication: whole files, partial files, fixed size blocks or variable sized blocks; (2) time when the deduplication occurs: inline during the write phase or as a background process; (3) precision of duplicate identification: can the system reliably find all duplicates or does it use an approximate technique which trades precision for increased performance? (4) the verification of equality between a duplicate and its copy: just by comparison of hashes or with full data comparison; (5) the scope of the deduplication: the whole system (global deduplication), or the deduplication limited for example to data on a specific node (local deduplication).

Today HYDRAstor implements variable-sized block, inline, hash-verified global duplicate elimination implemented on storage nodes. Variable-size blocks allow for better deduplication, because content-dependent chunking can be used ([40]). Inline deduplication increases write throughput, since duplicated block writes can be handled without writing to disk; this also increases storage efficiency compared to off-line deduplication. For regular blocks, we use fast approximate deduplication, whereas for retention roots, we do reliable duplicate elimination to ensure that searchable retention roots with the same search key but different contents are not written.

In both cases, for successful deduplication, we require that the potential duplicate of the block being written has a redundancy class not weaker than the class requested by this write and that the potential old duplicate is reconstructible.

On a regular block write, the peer handling this write is selected based on the hash of this block. It means that two identical blocks written when this peer is alive will be handled by it, and the second block will be found a duplicate of the first one.

A more complicated case arises when the write-handling peer has been recently created because of transfer or component recovery, and it does not have yet all the data it should have, i.e. its local SCC chain is not complete. In this case, we go to the longest-alive peer in the current supernode to check for possible duplicates. This is just a heuristics, as this peer may also not have the proper SCC chain complete, so a duplicate may not be detected. However, such a miss occurs only in corner cases, after massive failures when most likely all chains are broken. Moreover, for a particular block, we miss only one opportunity to eliminate a duplicate; the next duplicate block will be deduplicated unless another failure or transfer of this peer happens.

For retention roots, we need to ensure that two blocks with the same search key have identical contents (otherwise retention roots would not uniquely identify snapshots). As a result, we need accurate duplicate elimination for retention roots. When a local SCC chain has holes at the peer handling this write, the peer sends duplicate elimination queries to all other peers in this supernode. Each of these peers checks locally for a duplicate. A negative answer also includes a summary description of the parts of the SCC chain on which this answer is based. The write handling peer collects all replies. If there is at least one positive, a duplicate is found; otherwise, when all are negative, this peer tries to determine if SCC information attached to negative replies covers one entire SCC chain. If yes, the new block is not a duplicate; otherwise such determination cannot be done and the write is rejected with special error status indicating that data rebuilding is in progress (this may happen after massive failures); in such case this write should be submitted later. Needless to say, such situations so far happened only in special tests, and never in practice.

5.3 Deletion and Space Reclamation

Implementing data deletion in a system like HYDRAstor turned out to be surprisingly difficult because of many challenges which stem from the nature of the system: content-addressability, distribution, failure tolerance, and duplicate elimination. While deletion in our content-addressable system is somehow similar to distributed garbage collection [29], which is well understood, overcoming the remaining challenges, discussed below, required new research.

When deciding if a block is to be duplicate-eliminated against its older copy, we must be sure that this old block is not scheduled for deletion. Deciding which block to keep and which to delete must be globally consistent and robust in the presence of failures. For example, a deletion decision made should not be temporarily lost due to intermittent failures, as otherwise we may eliminate duplicates using blocks which are really scheduled for deletion. Moreover, the robustness of the data deletion algorithm should be higher than the data robustness. As a result, even if some blocks are lost, data deletion should be able to proceed to logically remove the lost data and heal the system if requested to do so by the user.

To simplify the design and make the implementation manageable, we have implemented deletion in two phases. During the first phase, the system is read-only and blocks are marked for deletion. In the second phase, the data can be read and written, as the system reclaims the blocks marked for deletion. Having a read-only phase simplified the deletion implementation, because such approach lets us eliminate the impact of writes on marking blocks for removal.

Deletion is implemented with a per-block reference counter that counts the number of pointers in blocks in the system pointing to this block. Reference counters are not updated immediately on write. Instead, they are updated later in the read-only phase processing all pointers written since the previous read-only phase (so the counter update is incremental). For each such pointer, the reference counter of the pointed block is incremented. After all such incrementation is completed, all blocks with reference counter equal to zero are marked for deletion (dark-shaded fragments in Fig. 3). Moreover, reference counters of blocks pointed by blocks already marked for deletion (including roots with associated deletion roots) are decremented. Next, the whole decrementation process (i.e. marking for removal blocks with reference counters equal to zero and decrementing reference counters of blocks pointed by pointers included in these blocks) is repeated, until no more new blocks can be marked for deletion. At this point, the read-only phase ends, and blocks marked for deletion can be removed in the background.

The deletion algorithm described above requires that the metadata of all blocks, as well as all the pointers, be present before proceeding. The pointers and block metadata are replicated on all peers, so the deletion can proceed even if some blocks are no longer reconstructible, as long as at least one block fragment exists.

Since blocks are really kept as fragments, a copy of the block reference counter is kept per-fragment, and each fragment of a given block should have the same value of this counter. Reference counters are computed independently on peers participating in the read-only phase. Before deletion is started, each such peer must have its SCC chain complete with respect to fragment metadata and pointers. Not all peers in a supernode have to participate, but some minimal number of peers is required to complete the read-only phase. The computed counters are later propagated in the background to remaining peers.

The redundancy in counter computation allows a deletion decision to survive node failures. However, the intermediate results of deletion computations are not persistent. Any failure before the decision is made wipes out these results on the affected nodes, and the whole computation needs to be repeated if too many peers cannot participate in this phase any more. Deletion can still continue, if a sufficient number of peers in each supernode are not affected by the failure. Upon conclusion of the read-only phase, the new counter values are made failure-tolerant. All dead blocks i.e. blocks with counters equal to zero are then swept out from physical storage in the background (reclamation in Fig. 3). Free space fragmentation is avoided by rewriting the whole synchrun component container, copying only fragments of live blocks to the new location.

6 Evaluation

Each current HYDRAstor storage node (SN) runs one back-end server, and has six 500 GB SATA disks, 6GB RAM, two dual-core 3 GHz CPUs and two GigE cards. Some experiments have also been done with the experimental next generation hardware (denoted SN2), in which each storage node runs two back-end servers and has twelve 1 TB SATA disks, 20 GB of RAM, two quad-core 3GHz CPUs and four GigE cards. In all experiments, we used the current access node (AN) with 6GB RAM, two dual-core 3 GHz CPUs, two GigE cards and only limited local storage. All nodes run the Red Hat EL 5.1 version of Linux.

All experiments were performed using block size of 64KB compressible by 33% to 48KB except where noted. The system was configured with a supernode cardinality of 12 and the number of supernodes was equal to the number of physical machines. All of the tests wrote data using a resiliency class which has 9 original and 3 redundant fragments.

6.1 Read/Write Bandwidth

This experiment shows write throughput as a function of the fraction of blocks detected as duplicates for two different compression ratios. We have used 4 *SN2* machines, and 4 *AN* machines, each with one testing driver able to generate a stream of blocks with a specified percentage of duplicates and compression ratio. Duplicates are evenly distributed in the stream. Duplicated data is written in the same order as the base data, re-creating the original data stream. For the read experiment, the testing driver attempts to read data in the same order as it was written.

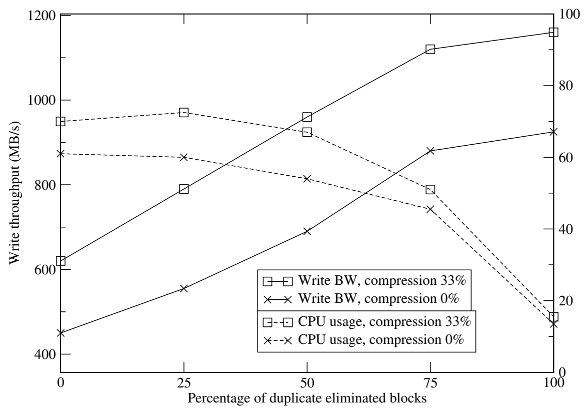

Figure 4: Write throughput as a function of duplicate ratio.

As shown in Fig. 4, very high bandwidth is achieved, which is a consequence of a carefully chosen data organization utilizing bulk transfer to disk. Duplicates are processed much more effectively than non-duplicated data, because they do not require fragmentation, fragment distribution and storage. Moreover, SCC-based organization allows the write-handling peer to perform fast local duplicate elimination by checking block reconstructibility with SCC reports submitted in the background from the remaining peers. However, when all writes are duplicates, the network bandwidth between AN and SNs becomes a bottle-neck, and the overall performance does not increase as much as expected (both curves flatten a bit at 100% duplicates). For high deduplication ratios, the CPU utilization decreases dramatically and network bandwidth between storage nodes remains available, so background tasks like data reconstruction and data scrubbing can be run without impact on user-visible performance.

Read bandwidth highly depends on factors like the sequentiality of the data read, the number of drivers reading simultaneously and the granularity of the distribution of the duplicates in the data. A detailed discussion of the impact of these factors on read performance is beyond the scope of this paper. Instead, we give read throughput achieved when reading the data written during the experiment described above. With four drivers reading, the total combined read bandwidth for indicated levels of deduplication was between 450 MB/s and 790MB/s for 33% compressible data, and between 400 MB/s and 550MB/s for 0% compressible data.

The time required to fill the 4 SN2 node system depends on the percentage of duplicates in the data written. The system can be filled in 1 day when writing with no duplicates, while filling a system with 95% duplicated data can take up to 10 days. In general, for configurations in which high performance is not a priority, fewer ANs can be used resulting also in extended time-to-fill.

These results were obtained with testing drivers running on the ANs. Experiments with real backup applications using the filesystem front-end yielded similar performance. However, since the experiments were not done in a controlled setup, their results are not presented here.

6.2 System Scaling

This experiment, with up to 12 *SNs* and the number of *AN*s set to half of the number of SNs, shows how performance is scaled when numbers of storage nodes and access nodes increase. Two sets of measurements are done — a dynamic one, in which nodes are added while the user is writing, and a static one in which the number of nodes remains constant during the test. In the latter case, each measurement was taken after re-initializing the system from scratch and then loading the same amount of random, non-duplicated data. Time on the X axis refers to the dynamic case only.

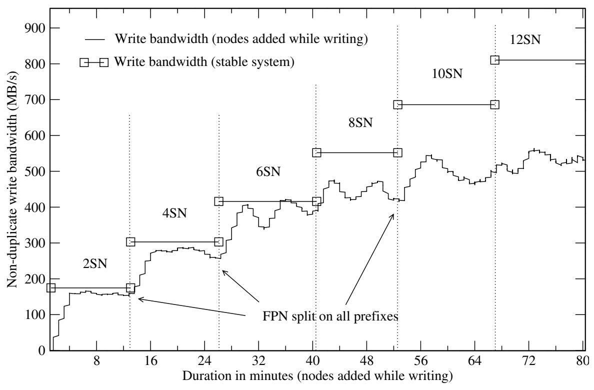

Figure 5: Dynamic vs. static scalability test.

The results indicate that in the range of nodes tested the system performance scales linearly with the system growth in the static case. The system attempts to balance components so that the hash space is divided equally across storage nodes. Such balancing guarantees that every machine is equally loaded and does not become a bottleneck. In the dynamic case, the cost of dynamically reconfiguring the system results in lower user bandwidth. This happens since most of the data is on the older nodes which are checked on every write for duplicate elimination. However, after all data transfers are completed, the performance in the dynamic case will be the same as in the stable case.

6.3 Node Failure and Data Rebuilding

This experiment shows the system behavior and its performance just after node failure, during resulting data reconstruction, and after the failed node is recovered. The system tested has 4 *SN2* machines and 4 *AN* machines.

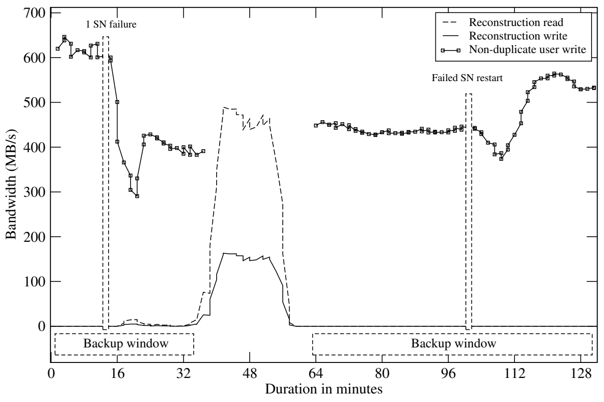

Figure 6: Node failure during backup.

We started writing to the healthy system with four storage nodes, achieving write throughput over 600 MB/s. After about 14 minutes one storage node failed (both back-end servers crashed). Write performance just after the node failure dropped to 300 MB/s, then stabilized at about 400 MB/s. The initial drop was caused by timed-out messages to the failed node and overhead for system rebalancing. Data rebuilding (reconstruction) tasks were ordered, however they were suppressed because of the ongoing user backup. Reconstruction started to work with full bandwidth just after all user writes had been finished. Every block reconstruction required reading all 9 remaining fragments in order to rebuild the 3 lost ones. The reconstruction read bandwidth reached 480 MB/s on the 3 surviving machines with a reconstruction write bandwidth of 160 MB/s. The rebuilding finished in the 58th minute of the experiment leaving a healthy system with only 3 storage nodes.

In the 64th minute the next writing session started achieving write bandwidth of 430 MB/s. The failed node was recovered and connected once again in the 100th minute. Just after the re-connection, system write bandwidth dropped to 380 MB/s, but when components rebalancing was finished it increased to about 550 MB/s. At the end of the experiment the system had 4 storage nodes, however it was not healthy, as not all data (SCCs) were in the correct places. Write performance will increase to the initial (600 MB/s) after all pending transfers are finished and the system becomes healthy again.

The results show that the system maximizes user bandwidth during backup even if background tasks are pending. In particular, ongoing reconstruction is suspended if a new backup is started. This approach allows a user to minimize costly backup windows regardless of internal system state, but carries the risk of starvation of critical data rebuilding tasks. However, this may happen only if the system is fully loaded by a user all the time and only when the user writes non-duplicated data. If the user load decreases or some duplicates are written, reconstruction is executed in the background. Finally, this experiment also shows how quickly the system adjusts to changes in its environment, as it takes only a few minutes for the system to fully utilize released resources.

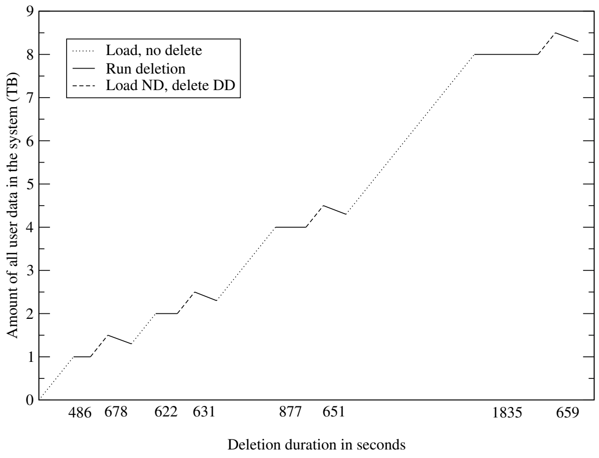

Figure 7: Read-only phase duration.

6.4 Data Deletion

The purpose of this experiment is to evaluate the duration of the read-only phase as a function of the data loaded with a focus on scenarios reflecting a realistic system usage. Therefore, we use the file system interface to write and delete data periodically, increasing overall the amount of data stored in the system. Deletion experiments were performed using 4 *SN2* machines and 1 *AN* machine.

The test runs the following four step sequence four times. During the initial step (shown with the dotted lines in the Fig. 7), the data is loaded into the system. The loading phase pauses at 1TB, 2TB, 4TB and 8TB. In the second step, during each pause, the read-only data deletion phase is run which recomputes the counters for the newly loaded data (note that no data is marked for deletion at this point). The duration of this step is shown with light-gray bars. The third step (shown with dashed lines), an additional half terabyte of new data (ND) is loaded and a user invokes a deletion operation of 0.2 TB of older data (DD). In the last step, one more read-only phase is run to recompute the counters to reflect the recently loaded data and mark the blocks for deletion.

The duration of each read-only phase is shown with dark-gray bars. In all cases, the new data is not compressible and does not contain any duplicates, but with duplicates present the results will be similar, except that all phases will be shorter.

Although the X axis in Fig. 7 shows duration of each read-only phase, the data-loading steps are not shown in proportion there, because they are too big (we load terabytes of data and it takes several hours). We note that all read-only phases are relatively short, the longest one, after loading 3.7 TB of data (which took about 4 hours) is about 30 minutes, resulting in deletion time of under 13% of writing time. For writing with two ANs, this fraction can go up to 20% in case of not-duplicated streams. When writing data with a high number of duplicates (the common case with backups), deletion takes significantly less time (on the order of 5% of the writing time), since less data needs to be read to access the pointers, and filling the capacity takes so much longer. Moreover, the duration of the first read-only phase (shown with the light-gray bars) in each sequence is proportional to the new data loaded in the first step of the scenario. Finally, the duration of the second read-only phase (shown with dark bars) is fairly constant, taking around 11 minutes per run. This also shows the power of the incremental reference-counting deletion in HYDRAstor. The duration of the read-only phase depends only on the amount of data added and deleted since the previous run of this phase, but not on the total amount of data in the system.

7 Related Work

A significant number of distributed storage systems [8, 10, 11] are designed as large scale systems which are distributed over wide area networks and built with untrusted peers. These systems undergo frequent configuration changes. For example, the goal of OceanStore [8] was to provide reliable storage for all data ever created. These systems concentrated on scalability (e.g. OceanStore, PAST [11]) and tolerating a large class of failures, including Byzantine and large-scale correlated failures (Glacier [18]) at the expense of performance.

Another group of distributed storage systems targeted the data center and, in this, are more like HYDRAstor. These systems include distributed virtual disk like Petal [19], distributed file systems like CEPH [37] and Farsite [6], clustered file systems like Sorrento [34], Panasas [39], and GoogleFS [15], clustered storage including Ursa Minor [5], RADOS [38], and FAB [28]. Compared to HYDRAstor these systems have different target applications and are not advertised as secondary storage. As a result, they do not provide deduplication (except Farsite, which does it on file level); these systems are not CAS-based, but need to deal with issues of consistency in the presence of write-sharing, which do not occur in our system. Ursa Minor does support user-selected choices of data resiliency, similar to our data resiliency classes. DISP [14] is a flexible system that can be specialized to both WAN and data center. Like HYDRAstor, DISP uses erasure codes, but it does not provide deduplication.

Venti [24], EMC Centera [4, 16], Pergamum [32] and DataDomain [43] are secondary storage systems. Venti, Pergamum and Centera target archiving, whereas DataDomain is designed to store backup data. Pergamum does not support duplicate elimination, Venti prototype and Centera do it, respectively, on fixed block size and entire file level. Centera might be able to do chunk-level deduplication, based on available information [16], but the chunk size seems to be much larger (100MB per chunk versus the HYDRAstor 64KB). These approaches result in lower deduplication than a variable-block size approach used by HYDRAstor and DataDomain. However, DataDomain is a centralized system and does not do global deduplication in distributed environment. HYDRAstor provides global deduplication using also variable block chunking with comparable write performance. RepStore [41], a smart-brick scalable storage system, uses erasure codes and content-based addressing, but does not provide deduplication. Deep Store [40], an archiving system, employs multitude of techniques for reducing stored data size, including delta compression and variable-block-size deduplication. However, this system does not target backup data.

Blocks in our system have some resemblance to objects in the object-based storage [22], as they have attributes (for example resiliency class) and simple interface to access its components like pointers.

Many systems introduce structures similar to SCCs for block aggregation. Venti uses arenas to serve as a unit of data maintenance; however, they do not take advantage of the sequential nature of incoming data streams and achieve very low performance. The Foundation [26] CAS Layer improves Venti's sequential write performance by prefetching entire arenas when duplicates are written. However, since Foundation is designed for personal use, it does not have to deal with the problem of multiple streams written concurrently but later read separately. DataDomain introduces containers to group sequential writes from each stream of data to increase effectiveness of read-ahead caching. HYDRAstor achieves a similar result by sorting incoming blocks by their stream id and flushing them out to disk in batches. Using separate containers for every stream in HYDRAstor is not feasible, as the number of containers written concurrently may be very large for big systems. HYDRAstor data organization is unique in use of replicated chains of containers which allow for reasoning about state of the data in the system.

Deletion in a distributed storage system is relatively simple if there is no duplicate elimination. It can be done with leases like in Glacier [18], or with simple reclamation of obsolete versions like in Ursa Minor. However, with deduplication, deletion becomes difficult for reasons explained earlier. For example, Venti and Deep Store have not implemented deletion. As far as we know, the HYDRAstor back-end approach to deletion is unique. The use of blocks with pointers, retention and deletion roots and redundant chains of containers enables an efficient, fault-tolerant implementation of a distributed deletion.

8 Conclusions and Future Work

HYDRAstor is a decentralized, scalable secondary storage that is commercially available today. It can be used as an on-line repository for all enterprise backup and archival data while dynamically and efficiently sharing available capacity. Critical features like high-availability and reliability, ease of management, capacity and performance scalability, and storage efficiency make the system unique in addressing today's enterprise needs. The system is externally visible as one storage pool and can be accessed by legacy applications using traditional file system interface.

The core architecture is built around a DHT with virtual supernodes spanned over physical nodes. Data resiliency is provided with erasure codes, with fragments of erasure-coded blocks distributed among supernode components. Redundancy in the network and data allows for on-line upgrades and extensions, increasing availability of the system. High storage efficiency is facilitated by variable block size global deduplication. The back-end exports a low-level API providing operations on content-addressed blocks which expose pointers to other blocks. A novel data organization based on redundant chains of data containers is used to deliver reliably multitude of data services, including failure-tolerant deletion and fast verification of data health.

Although the system is fully functional today, there is an important work left to improve its value delivered to the end user. The read-only phase of deletion will be eliminated, which will make the system fully usable all the time. Deduplication can be moved to a proxy server, saving bandwidth and improving write performance of highly-duplicated streams. Additionally, since multiple types of drivers can write to the back-end, there is a need for a stream interface that can cut data into blocks in a standard way. This will ensure higher deduplication among data written by different types of clients.

References

[1] Plasmon and Pegasus. Archival Storage Total Cost of Ownership Analysis, 2005. http://www.pegasus-afs.com/PDFs/White Papers/Archive Storage TCO Report.pdf.

[2] Vtf open, 2005. http://www.diligent.com/products:VTF-Open-2.

[3] Overland storage unveils reo 9500d all-in-one deduplicating vtl appliance, 2007. http://www.overlandstorage.com.

[4] EMC Corp. EMC Centera: content addressed storage system, 2008. http://www.emc.com/products/family/emc-centera-family.htm?-openfolder=platform.

[5] ABD-EL-MALEK, M., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M. P., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa minor: Versatile cluster-based storage. In *FAST* (2005).

[6] ADYA, A., BOLOSKY, W., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. Farsite: Federated, available, and reliable storage for an incompletely trusted environment, 2002.

[7] AJMANI, S., LISKOV, B., AND SHRIRA, L. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)* (July 2006).

[8] BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., ZHAO, B., AND KUBIATOWICZ, J. Oceanstore: An extremely wide-area storage system. Tech. rep., Berkeley, CA, USA, 1999.

[9] BLÖMER, J., KALFANE, M., KARPINSKI, M., KARP, R., LUBY, M., AND ZUCKERMAN, D. An xor-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, August 1995.

[10] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with cfs. *SIGOPS Oper. Syst. Rev. 35*, 5 (2001), 202–215.

[11] DRUSCHEL, P., AND ROWSTRON, A. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII* (Schloss Elmau, Germany, May 2001), pp. 75–80.

[12] DUBNICKI, C., UNGUREANU, C., AND KILIAN, W. FPN: A Distributed Hash Table for Commercial Applications. In *Proceedings of the Thirteenth International Symposium on High-Performance Distributed Computing (HPDC-13 2004)* (Honolulu, Hawaii, June 2004), pp. 120–128.

[13] EL MALEK, M. A., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M., PRASAD, M., SALMON, O., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Early experiences on the journey towards self-* storage. *IEEE Data Eng. Bulletin* (2006).

[14] ELLARD, D., AND MEGQUIER, J. Disp: Practical, efficient, secure and fault-tolerant distributed data storage. *Trans. Storage 1*, 1 (2005), 71–94.

[15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[16] GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. Deconstructing Commodity Storage Clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture* (Madison, WI, June 2005).

[17] GUPTA, S. D. Network Magazine. Addressing Storage Management Challenges, 2002. http://www.networkmagazineindia.com/200212/cover1.shtml.

[18] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 143–158.

[19] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1996), pp. 84–92.

[20] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In *In Proceedings of IPTPS02* (Cambridge, USA, March 2002).

[21] MERRILL, D. Hitachi Data Systes. Storage Economics: Identifying and Reducing Operating Expenses in the Storage Infrastructure , 2003. http://www.hds.com/pdf/StorageEconomicsWHP-153.pdf.

[22] MESNIER, M., GANGER, G. R., AND RIEDEL, E. Object-based storage. *IEEE Communications Magazine 41* (2003), 84–90.

[23] NEC Corporation. HYDRAstor Grid Storage System, 2008. http://www.hydrastor.com.

[24] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies* (Monterey,CA, 2002).

[25] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2001), ACM, pp. 161–172.

[26] RHEA, S., COX, R., AND PESTEREV, A. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 143–156.

[27] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science 2218* (2001), 329+.

[28] SAITO, Y., FROLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. Fab: building distributed enterprise disk arrays from commodity components. *SIGOPS Oper. Syst. Rev. 38*, 5 (2004), 48–58.

[29] SHAPIRO, M. A survey of distributed garbage collection techniques. In *In Proceedings of the 1995 International Workshop on Memory Management* (1995), Springer-Verlag, pp. 211–249.

[30] SOULES, C. A. N., APPAVOO, J., HUI, K., WISNIEWSKI, R. W., SILVA, D. D., GANGER, G. R., KRIEGER, O., STUMM, M., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., AND XENIDIS, J. System support for online reconfiguration, June 2003.

[31] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications* (2001), ACM Press, pp. 149–160.

[32] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–16.

[33] STRUNK, J. D., AND GANGER, G. R. A human organization analogy for self-* systems. In *In First Workshop on Algorithms and Architectures for Self-Managing Systems, in conjunction with Federated Computing Research Conference* (2003), pp. 1–6.

[34] TANG, H., GULBEDEN, A., ZHOU, J., STRATHEARN, W., YANG, T., AND CHU, L. A self-organizing storage cluster for parallel data-intensive applications. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2004), IEEE Computer Society, p. 52.

[35] THERESKA, E., SALMON, O., STRUNK, J., WACHS, M., EL MALEK, M. A., LOPEZ, J., AND GANGER, G. R. Stardust: Tracking activity in a distributed storage system. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Saint-Malo* (2006), ACM Press, pp. 3–14.

[36] WEATHERSPOON, H., AND KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems* (London, UK, 2002), Springer-Verlag, pp. 328–338.

[37] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *OSDI'06: 7th USENIX Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 307–320.

[38] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *PDSW* (2007), G. A. Gibson, Ed., ACM Press, pp. 35–44.

[39] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., AND MUELLER, B. Scalable performance of the panasas parallel file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 17–33.

[40] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep store: An archival storage system architecture. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 804–8015.

[41] ZHANG, Z., LIN, S., LIAN, Q., AND JIN, C. Repstore: A self-managing and self-tuning storage backend with smart bricks. In *ICAC* (2004), pp. 122–129.

[42] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22* (2004), 41–53.

[43] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–14.