SafeStore: A Durable and Practical Storage System

Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin
The University of Texas at Austin

Abstract

This paper presents SafeStore, a distributed storage system designed to maintain long-term data durability despite conventional hardware and software faults, environmental disruptions, and administrative failures caused by human error or malice. The architecture of SafeStore is based on fault isolation*, which SafeStore applies aggressively along administrative, physical, and temporal dimensions by spreading data across autonomous storage service providers (SSPs). However, current storage interfaces provided by SSPs are not designed for high end-to-end durability. In this paper, we propose a new storage system architecture that (1) spreads data efficiently across autonomous SSPs using* informed hierarchical erasure coding *that, for a given replication cost, provides several additional 9's of durability over what can be achieved with existing black-box SSP interfaces, (2) performs an efficient end-to-end audit of SSPs to detect data loss that, for a 20% cost increase, improves data durability by two 9's by reducing MTTR, and (3) offers durable storage with cost, performance, and availability competitive with traditional storage systems. We instantiate and evaluate these ideas by building a SafeStore-based file system with an NFS-like interface.*

1 Introduction

The design of storage systems that provide data durability on the time scale of decades is an increasingly important challenge as more valuable information is stored digitally [10, 31, 60]. For example, data from the National Archives and Records Administration indicate that 93% of companies go bankrupt within a year if they lose their data center in some disaster [5], and a growing number of government laws [8, 22] mandate multi-year periods of data retention for many types of information [12, 51].

Against a backdrop in which over 34% of companies fail to test their tape backups [6] and over 40% of individuals do not back up their data at all [29], multi-decade scale durable storage raises two technical challenges. First, there exist a broad range of threats to data durability including media failures [52, 63, 70], software bugs [53, 71], malware [18, 66], user error [51, 62], administrator error [40, 49], organizational failures [24, 28], malicious insiders [27, 32], and natural disasters on the scale of buildings [7] or geographic regions [11]. Requiring robustness on the scale of decades magnifies them all: threats that could otherwise be considered negligible must now be addressed. Second, such a system has to be practical with cost, performance, and availability competitive with traditional systems.

Storage outsourcing is emerging as a popular approach to address some of these challenges [42]. By entrusting storage management to a Storage Service Provider (SSP), where "economies of scale" can minimize hardware and administrative costs, individual users and small to medium-sized businesses seek cost-effective professional system management and peace of mind vis-a-vis both conventional media failures and catastrophic events.

Unfortunately, relying on an SSP is no panacea for long-term data integrity. SSPs face the same list of hard problems outlined above and as a result even brand-name ones [9, 14] can still lose data. To make matters worse, clients often become aware of such losses only after it is too late. This opaqueness is a symptom of a fundamental problem: SSPs are separate administrative entities and the internal details of their operation may not be known by data owners. While most SSPs may be highly competent and follow best practices punctiliously, some may not. By entrusting their data to back-box SSPs, data owners may free themselves from the daily worries of storage management, but they also relinquish ultimate control over the fate of their data. In short, while SSPs are an economically attractive response to the costs and complexity of long-term data storage, they do not offer their clients any end-to-end guarantees on data durability, which we define as the probability that a specific data object will not be lost or corrupted over a given time period.

To achieve high durability, SafeStore applies aggressively the principle of *fault isolation* without compromising practicality in terms of cost, performance, and availability.

Aggressive isolation for durability. SafeStore stores data redundantly across multiple SSPs and leverages diversity across SSPs to prevent permanent data loss caused by isolated administrator errors, software bugs, insider attacks, bankruptcy, or natural catastrophes. With respect to data stored at each SSP, SafeStore employs a "trust but verify" approach: it does not interfere with the policies used within each SSP to maintain data integrity, but it provides an *audit* interface so that data owner retain end-to-end control over data integrity. The audit mechanism can quickly detect data loss and trigger data recovery from redundant storage before additional faults result in unrecoverable loss. Finally, to guard data stored at SSPs against faults at the data owner site (e.g. operator errors, software bugs, and malware attacks), SafeStore restricts the interface to provide temporal isolation between clients and SSPs so that the latter export the abstraction of write-once-read-many storage.

Making aggressive isolation practical. SafeStore introduces an efficient storage interface to reduce network bandwidth and storage cost using an *informed hierarchical erasure coding* scheme, that, when applied across and within SSPs, can achieve near-optimal durability. SafeStore SSPs expose redundant encoding options to allow the system to efficiently divide storage redundancies across and within SSPs. Additionally, SafeStore limits the cost of implementing its "trust but verify" policy through an audit protocol that shifts most of the processing to the audited SSPs and encourages them proactively measure and report any data loss they experience. Dishonest SSPs are quickly caught with high probability and at little cost to the auditor using probabilistic spot checks. Finally, to reduce the bandwidth, performance, and availability costs of implementing geographic and administrative isolation, SafeStore implements a two-level storage architecture where a local server (possibly running on the client machine) is used as a soft-state cache, and if the local server crashes, SafeStore limits down-time by quickly recovering the critical meta data from the remote SSPs while the actual data is being recovered in the background.

Contributions. The contribution of this paper is a highly durable storage architecture that uses a new replication interface to distribute data efficiently across diverse set of SSPs and an effective audit protocol to check data integrity. We demonstrate that this approach can provide high durability in a way that is practical and economically viable with cost, availability, and performance competitive with traditional systems. We demonstrate these ideas by building and evaluating SSFS, an NFS-based SafeStore storage system. Overall, we show that SafeStore provides an economical alternative to realize multi-decade scale durable storage for individuals and small-to-medium sized businesses with limited resources. Note that although we focus our attention on outsourced SSPs, the SafeStore architecture could also be applied internally by large enterprises that maintain multiple isolated data centers.

2 Architecture and Design Principles

The main goal of SafeStore is to provide extremely durable storage over many years or decades.

2.1 Threat model

Over such long time periods, even relatively rare events can affect data durability, so we must consider broad range of threats along multiple dimensions—physical, administrative, and software.

Physical faults: Physical faults causing data loss include disk media faults [35, 70], theft [23], fire [7], and wider geographical catastrophes [11]. These faults can result in data loss at a single node or spanning multiple nodes at a site or in a region.

Administrative and client-side faults: Accidental misconfiguration by system administrators [40, 49], deliberate insider sabotage [27, 32], or business failures leading to bankruptcy [24] can lead to data corruption or loss. Clients can also delete data accidentally by, for example, executing "rm -r *". Administrator and client faults can be particularly devastating because they can affect replicas across otherwise isolated subsystems. For instance [27], a system administrator not only deleted data but also stole the only backup tape after he was fired, resulting in financial damages in excess of $10 million and layoff of 80 employees.

Software faults: Software bugs [53, 71] in file systems, viruses [18], worms [66], and Trojan horses can delete or corrupt data. A vivid example of threats due to malware is the recent phenomenon of ransomware [20] where an attacker encrypts a user's data and withholds the encryption key until a ransom is paid.

Of course, any of the listed faults may occur rarely. But at the scale of decades, it becomes risky to assume that no rare events will occur. It is important to note that some of these failures [7, 52, 63] are often correlated resulting in simultaneous data loss at multiple nodes while others [53] are more likely to occur independently.

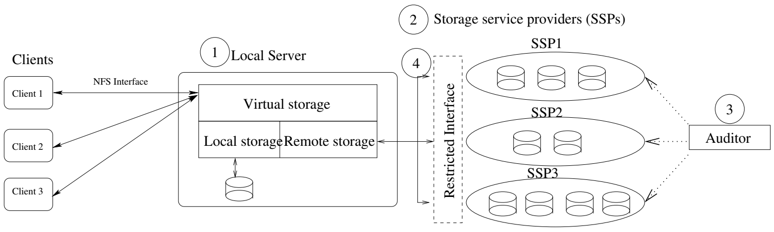


Fig. 1: SafeStore architecture

Replication mechanisms optimized for one or the other type of failure may not be optimal in this setting where both failure types can happen.

Limitations of existing practice. Most existing approaches to data storage face two problems that are particularly acute in our target environments of individuals and small/medium businesses: (1) they depend too heavily on the operator or (2) they provide insufficient fault isolation in at least some dimensions.

For example, traditional removable-media-based-systems (e.g., tape, DVD-R) systems are labor intensive, which hurts durability in the target environments because users frequently fail to back their data up, fail to transport media off-site, or commit errors in the backup/restore process [25]. The relatively high risk of robot and media failures [3] and slow mean time to recover [45] are also limitations.

Similarly, although on-site disk-based [4, 16] backup systems speed backup/recovery, use reliable media compared to tapes, and even isolate client failures by maintaining multiple versions of data, they are vulnerable to physical site, administrative, and software failures.

Finally, network storage service providers (SSPs) [1, 2, 15, 21] are a promising alternative as they provide geographical and administrative isolation from users and they ride the technology trend of falling network and hardware costs to reduce the data-owner's effort. But they are still vulnerable to administrative failures at the service providers [9], organizational failures (e.g., bankruptcy [24, 42]), and operator errors [28]. They thus fail to fully meet the challenges of a durable storage system. We do, however, make use of SSPs as a component of SafeStore.

2.2 SafeStore architecture

As shown in Figure 1, SafeStore uses the following design principles to provide high durability by tolerating the broad range of threats outlined above while keeping the architecture practical, with cost, performance, and availability competitive with traditional systems.

Efficiency via 2-level architecture. SafeStore uses a two-level architecture in which the data owner's *local server* (1 in Figure 1) acts as a cache and write buffer while durable storage is provided by multiple remote *storage service providers* SSPs 2. The local server could be running on the client's machine or a different machine. This division of labor has two consequences. First, performance, availability, and network cost are improved because most accesses are served locally; we show this is crucial in Section 3. Second, management cost is improved because the requirements on the local system are limited (local storage is soft state, so local failures have limited consequences) and critical management challenges are shifted to the SSPs, which can have excellent economies of scale for managing large data storage systems [1, 26, 42].

Aggressive isolation for durability. We apply the principle of aggressive isolation in order to protect data from the broad range of threats described above.

- *Autonomous SSPs:* SafeStore stores data redundantly across multiple autonomous SSPs (2 in Figure 1). Diverse SSPs are chosen to minimize the likelihood of common-mode failures across SSPs. For example, SSPs can be external commercial service providers [1, 2, 15, 21], that are geographically distributed, run by different companies, and based on different software stacks. Although we focus on *out-sourced* SSPs, large organizations can use our architecture with *in-sourced* storage across autonomous entities within their organization (e.g., different campuses in a university system.)
- *Audit:* Aggressive isolation alone is not enough to provide high durability as data fragment failures accumulate over time. On the contrary, aggressive isolation can adversely affect data durability because the data owner has little ability to enforce or monitor the SSPs' internal design or operation to ensure that SSPs follow best practices. We provide an end-to-end audit interface (3 in Figure 1) to detect data loss and thereby bound mean time to recover (MTTR), which in turn increases mean time to data loss (MTTDL). In Section 4 we describe our audit interface and show how audits limit the damage that poorly-run SSPs can inflict on overall durability.

- *Restricted interface:* SafeStore must minimize the likelihood that erroneous operation of one subsystem compromises the integrity of another [47]. In particular, because SSPs all interact with the local server, we must restrict that interface. For example, we must protect against careless users, malicious insiders, or devious malware at the clients or local server that mistakenly delete or modify data. SafeStore's restricted SSP interface ④ provides temporal isolation via the abstraction of versioned write-once-read-many storage so that a future error cannot damage existing data.

Making isolation practical. Although durability is our primary goal, the architecture must still be economically viable.

- *Efficient data replication:* The SafeStore architecture defines a new interface that allows the local server to realize near-optimal durability using *informed hierarchical erasure coding* mechanism, where SSPs expose internal redundancy. Our interface does not restrict SSP's autonomy in choosing internal storage organization (replication mechanism, redundancy level, hardware platform, software stack, administrative policies, geographic location, etc.) Section 3 shows that our new interface and replication mechanism provides orders of magnitude better durability than *oblivious hierarchical encoding* based systems using existing black-box based interfaces [1, 2, 21].
- *Efficient audit mechanism:* To make audits of SSPs practical, we use a novel audit protocol that, like real world financial audits, uses self-reporting whereby auditor offloads most of the audit work to the auditee (SSP) in order to reduce the overall system resources required for audits. However, our audit takes the form of a challenge-response protocol with occasional spot-checks that ensure that an auditee that generates improper responses is quickly discovered and that such a discovery is associated with a cryptographic proof of misbehavior [30].
- *Other optimizations:* We use several optimizations to reduce overhead and downtime in order to make system practical and economically viable. First, we use a fast recovery mechanism to quickly recover from data loss at a local server where the local server comes online as soon as the meta-data is recovered from remote SSPs even while data recovery is going on in the background. Second, we use block level versioning to reduce storage and network overhead involved in maintaining multiple versions of files.

2.3 Economic viability

In this section, we consider the economic viability of our storage system architecture in two different settings,

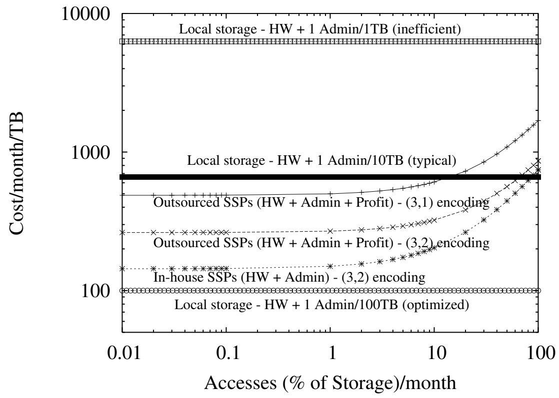


Fig. 2: Comparison of SafeStore cost v. accesses to remote storage (as a percentage of straw-man Standalone local storage) varies.

outsourced storage using commercial SSPs and federated storage using in-house but autonomous SSPs, and calibrate the costs by comparing with a less-durable local storage system.

We consider three components to storage cost: hardware resources, administration, and—for outsourced storage—profit. Table 1 summarizes our basic assumptions for a straw-man *Standalone* local storage system and for the local owner and SSP parts of a SafeStore system. In column B, we estimate the raw hardware and administrative costs that might be paid by an in-house SSP. We base our storage hardware costs on estimated full-system 5-year total cost of ownership (TCO) costs in 2006 for large-scale internet services such as Internet Archive [26]. Note that using the same storage cost for a large-scale, specialized SSP and for smaller data owners and Standalone systems is conservative in that it may overstate the relative additional cost of adding SSPs. For network resources, we base our costs on published rates in 2006 [17]. For administrative costs, we use Gray's estimate that highly efficient internet services require about 1 administrator to manage 100TB while smaller enterprises are typically closer to one administrator per 10TB but can range from one per 1TB to 1 per 100TB [50] (Gray notes, "But the real cost of storage is management" [50]). Note that we assume that by transforming local storage into a soft-state cache, SafeStore simplifies local storage administration. We therefore estimate local hardware and administrative costs at 1 admin per 100TB.

In Figure 2, the storage cost of in-house SSP includes SafeStore's hardware (cpu, storage, network) and administrative costs. We also plot the straw-man local storage system with 1, 10, or 100 TB per administrator. The outsourced SSP lines show SafeStore costs assuming SSPs prices include a profit by using Amazon's S3

	Standalone	SafeStore In-house	SafeStore SSP (Cost+Profit)
Storage	\$30/TB/month [26]	\$30/TB/month [26]	\$150/TB/month [1]
Network	NA	\$200/TB [17]	\$200/TB [1]
Admin	1 admin/[1,10,100]TB ([inefficient,typical,optimized]) [50]	1 admin/100TB [50]	Included [1]

Table 1: System cost assumptions. Note that a *Standalone* system makes no provision for isolated backup and is used for cost comparison only.

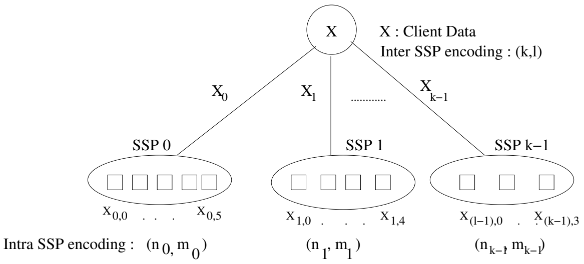


Fig. 3: Hierarchical encoding

storage service pricing. Three points stand out. First, additional replication to SSPs increases cost (as inter-SSP data encoding, as discussed in section 3, is raised from (3,2) to (3,1)), and the network cost rises rapidly as the remote access rate increases. These factors motivate SafeStore's architectural decisions to (1) use efficient encoding and (2) minimize network traffic with a large local cache that fully replicates all stored state. Second, when SSPs are able to exploit economies of scale to reduce administrative costs below those of their customers, SafeStore can reduce overall system costs even when compared to a less-durable Standalone local-storage-only system. Third, even for customers with highly-optimized administrative costs, as long as most requests are filtered by the local cache, SafeStore imposes relatively modest additional costs that may be acceptable if it succeeds in improving durability.

The rest of the paper is organized as follows. First, in Section 3 we present and and evaluate our novel *informed hierarchical erasure coding* mechanism. In Section 4, we address SafeStore's audit protocol. Later, in Section 5 we describe the SafeStore interfaces and implementation. We evaluate the prototype in Section 6. Finally, we present the related work in Section 7.

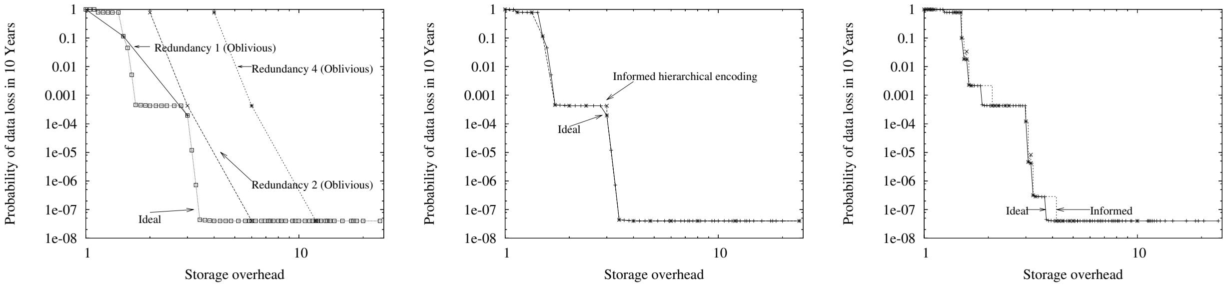
3 Data replication interface

This section describes a new replication interface to achieve near-optimal data durability while limiting the internal details exposed by SSPs, controlling replication cost, and maximizing fault isolation.

As shown in Figure 3, SafeStore uses hierarchical encoding comprising inter-SSP and intra-SSP redundancy: First, it stores data redundantly across different SSPs, and then each SSP internally replicates data entrusted to it as it sees fit. Hierarchical encoding is the natural way to replicate data in our setting as it tries to maximize fault-isolation across SSPs while allowing SSP's autonomy in choosing an appropriate internal data replication mechanism. Different replication mechanisms such as erasure coding [57], RAID [35], or full replication can be used to store data redundantly at inter-SSP and intra-SSP levels (any replication mechanism can be viewed as some form of (k,l) encoding [68] from durability perspective, where l out of k encoded fragments are required to reconstruct data). However, it requires proper balance between inter-SSP and intra-SSP redundancies to maximize end-end durability for a fixed storage overhead. For example, consider a system willing to pay an overall 6x redundancy cost using 3 SSPs with 8 nodes each. If, for example, each SSP only provides the option of (8,2) intra-SSP encoding, then we can use at most (3,2) inter-SSP encoding. This combination gives gives 4 9's less durability for the same overhead compared to a system that uses (3,1) encoding at the inter-SSP level and (8,4) encoding at the intra-SSP level at all the SSPs.

3.1 Model

The overall storage overhead to store a data object is $(n_0/m_0 + n_1/m_1 + ...n_{k-1}/m_{k-1})/l$, when a data object is hierarchically encoded (as shown in Figure 3) using (k,l) erasure coding across k SSPs, and SSPs 0 through $k-1$ internally use erasure codings (n_0, m_0), (n_1, m_1),....(n_{k-1}, m_{k-1}), respectively. We assume that the number of SSPs(k) is fixed and a data object is (possibly redundantly) stored at all SSPs. We do not allow varying k as it requires additional internal information about various SSPs (MTTF of nodes, number of nodes, etc.) which may not be available in order to choose optimal set of k nodes. Instead, we tackle the problem of finding optimal distribution of inter-SSP and intra-SSP redundancies for a fixed k. The end-to-end data durability can be estimated as a function of these variables using a simple analytical model, detailed in Appendix A of our extended report [46], that considers two classes of faults. *Node faults* (e.g. physical faults like sector failures, disk crashes, etc.) occur within an SSP and affect just one fragment of an encoded object stored at the SSP. *SSP faults* (e.g., administrator errors, organi-



(a) (b) (c)

Fig. 4: (a) Durability with Black-box interface with fixed intra-SSP redundancy (b) Informed hierarchical encoding (c) Informed hierarchical encoding with non-uniform distribution

zational failures, geographical failures, etc.) are instead simultaneous or near-simultaneous failures that take out all fragments across which an object is stored within an SSP. To illustrate the approach, we consider a baseline system consisting of 3 SSPs with 8 nodes each. We use a baseline MTTDL of 10 years due to invidual node faults and 100 years for SSP failures and assume both are independent and identically distributed. We use MTTR of data of 2 days (e.g. to detect and replace a faulty disk) for node faults and 10 days for SSP failures. We use the probability of data loss of an object during a 10 year period to characterize durability because expressing end-to-end durability as MTTDL can be misleading [35] (although MTTDL can be easily computed from the probability of data loss as shown in Appendix A. Later, we change the distribution of nodes across SSPs, MTTDL and MTTR of node failures within SSPs, to model diverse SSPs. The conclusions that we draw here are general and not specific to this setup; we find similar trends when we change the total number of nodes, as well as MTTDL and MTTR of correlated *SSP faults*.

3.2 Informed hierarchical encoding

A client can maximize end-to-end durability if it can control both intra-SSP and inter-SSP redundancies. However, current black-box storage interfaces exported by commercial outsourced SSPs [1, 2, 21] do not allow clients to change intra-SSP redundancies. With such a black-box interface, clients perform *oblivious hierarchical encoding* as they control only inter-SSP redundancy. Figure 4(a) plots the optimal durability achieved by an *ideal* system that has full control of inter-SSP and intra-SSP redundancy and a system using *oblivious hierarchical encoding*. The latter system has 3 lines for different fixed intra-SSP redundancies of 1, 2, and 4, where each line has 3 points for each of the 3 different inter-SSP encodings((3,1), (3,2) and (3,3)) that a client can choose with such a black-box interface. Two conclusions emerge. First, for a given storage overhead, the probability of data loss of an *ideal* system is often orders of magnitude lower than a system using *oblivious hierarchical encoding*, which therefore is several 9's short of optimal durability. Second, a system using *oblivious hierarchical encoding* often requires 2x-4x more storage than *ideal* to achieve the same durability.

To improve on this situation, SafeStore describes an interface that allows clients to realize near-optimal durability using *informed hierarchical encoding* by exercising additional control on intra-SSP redundancies. With this interface, each SSP exposes the set of redundancy factors that it is willing to support. For example, an SSP with 4 internal nodes can expose redundancy factors of 1 (no redundancy), 1.33, 2, and 4 corresponding, respectively, to the (4,4), (4,3), (4,2) and (4,1) encodings used internally.

Our approach to achieve near-optimal end-to-end durability is motivated by the stair-like shape of the curve tracking the durability of *ideal* as a function of storage overhead (Figure 4(a)). For a fixed storage overhead, there is a tradeoff between inter-SSP and intra-SSP redundancies, as a given overhead O can be expressed as $1/l \times (r_0 + r_1 + ..r_{k-1})$, when (k,l) encoding is used across k SSPs in the system with intra-SSP redundancies of r_0 to r_{k-1} (where $r_i = n_i/m_i$). Figure 4(a) shows that durability increases dramatically (moving down one step in the figure) when inter-SSP redundancy increases, but does not improve appreciably when additional storage is used to increase intra-SSP redundancy beyond a threshold that is close to but greater than 1. This observation is backed by mathematical analysis as explained in observation 1 of Appendix A.

Hence, we propose a heuristic biased in favor of spending storage to maximize inter-SSP redundancy as follows:

- First, for a given number k of SSPs, we maximize the inter-SSP redundancy factor by minimizing l. In particular, for each SSP i, we choose the minimum redundancy factor $r'_i > 1$ exposed by i, and we compute

l as $l = \lfloor (r'_0 + r'_1 + ...r'_{k-1})/O \rfloor$.

- Next, we distribute the remaining overhead $(O - 1/l \times (r'_0 + r'_1 + ..r'_{k-1}))$ among the SSPs to minimize the standard deviation of the intra-SSP redundancy factors r_i that are ultimately used by the different SSPs. We minimize standard deviation by initializing it to the lowest possible value (0%) and then distribute overhead across all intra-SSP redundancies so that the deviation is within the value and new intra-SSP redundancies are allowed by SSPs. If we do not find a possible set of allowable intra-SSP redundancies then we relax standard deviation constraint by increasing it gradually and follow the above step until we find an allowable set of intra-SSP redundancies.

The first rule is used to maximize inter-SSP redundancy and the second rule is to ensure that intra-SSP redundancies are uniformly distributed across SSPs. We try to distribute redundancy uniformly across all SSPs otherwise SSPs with small or no redundancy tend to loose objects faster and require expensive inter-SSP recovery to recover from such failures.

Figure 4(b) shows that this new approach, which we call *informed hierarchical coding*, achieves near optimal durability in a setting where three SSPs have the same number of nodes (8 each) and the same MTTDL and MTTR for internal node failures. These assumptions, however, may not hold in practice, as different SSPs are likely to have a different number of nodes, with different MTTDLs and MTTRs. Figure 4(c) shows the result of an experiment in which SSPs have a different number of nodes—and, therefore, expose different sets of redundancy factors. We still use 24 nodes, but we distribute them non-uniformly (14, 7, 3) across the SSPs: informed hierarchical encoding continues to provide near-optimal durability. This continues to be true even when there is a skew in MTTDL and MTTR (due to node failures) across SSPs. For instance, Figure 5 uses the same non-uniform node distribution of Figure 4(c), but the (MTTDL, MTTR) values for node failures now differ across SSPs—they are, respectively, (10 years, 2 days), (5 years, 3 days), and (3 years, 5 days). Note that, by assigning the worst (MTTDL, MTTR) for node failures to the SSP with least number of nodes, we are considering a worst-case scenario for informed hierarchical encoding.

These results are not surprising in light of our discussion of Figure 4(a): durability depends mainly on maximizing inter-SSP redundancy and it is only slightly affected by the internal data management of individual SSPs. In Appendix D we study the sensitivity of informed hierarchical encoding to changes in the total number of nodes used to store data across all SSPs and in MTTDL and MTTR for SSP failures: they all confirm the conclusion that a simple interface that allows SSPs to expose the redundancy factors they support is all it is needed to achieve, through our simple informed hierarchical encoding mechanism, near optimal durability.

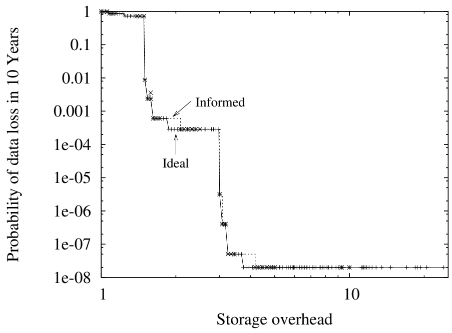


Fig. 5: Durability with different MTTDL and MTTR for node failures across SSPs

SSPs can provide such an interface as part of their SLA (service level agreement) and charge clients based on the redundancy factor they choose when they store a data object. The interface is designed to limit the amount of detail that an SSP must expose about the internal organization. For example, an SSP with 1000 servers each with 10 disks might only expose redundancy options (1.0, 1.1, 1.5, 2.0, 4.0, 10.0), revealing little about its architecture. Note that the proposed interface could allow a dishonest SSP to cheat the client by using less redundancy than advertised. The impact of such false advertising is limited by two factors: First, as observed above, our design is relatively insensitive to variations in intra-SSP redundancy. Second, the end to end audit protocol described in the next section limits the worst-case damage any SSP can inflict.

4 Audit

We need an effective audit mechanism to quickly detect data losses at SSPs so that data can be recovered before multiple component failures resulting in unrecoverable loss. An SSP *should* safeguard the data entrusted to it by following best practices like monitoring hardware health [65], spreading coded data across drives and controllers [35] or geographically distributed data centers, periodically scanning and correcting latent errors [64], and quickly notifying a data owner of any lost data so that the owner can restore the data from other SSPs and maintain a desired replication level. However, the principle of isolation argues against blindly assuming SSPs are flawless system designers and operators for two reasons. First, SSPs are separate administrative entities, and their internal details of operation may not be verifiable by data owners. Second, given the imperfections of software [18, 53, 71], operators [40, 49], and hardware [35, 70], even name-brand SSPs may encounter unexpected issues and silently lose customer data [9, 14]. Auditing SSP data storage embodies the end-to-end principle (in almost exactly the form it was first described) [61], and frequent auditing ensures a short Mean Time To Detect (MTTD) data loss, which helps limit worst-case Mean Time To Recover (MTTR). It is important to reduce MTTR in order to increase MTTDL as a good replication mechanism alone cannot improve MTTDL over a long time-duration spanning decades.

The technical challenge to auditing is to provide an end-to-end guarantee on data integrity while minimizing cost. These goals rule out simply reading stored data across the network as too expensive (see Figure 2) and, similarly, just retrieving a hash of the data as not providing an end-to-end guarantee (the SSP may be storing the hash not the data.). Furthermore, the audit protocol must work with data erasure-coded across SSPs, so a simple scheme that sends a challenge to multiple identical replicas and then compare the responses such as those in LOCKSS [47] and Samsara [37] do not work. We must therefore devise an inexpensive audit protocol despite the fact that no two replicas store the same data.

To reduce audit cost, SafeStore's audit protocol borrows a strategy from real-world audits: we push most of the work onto the auditee and ask the auditor to spot check the auditee's reports. Our reliance on self-reporting by SSPs drives two aspects of the protocol design. First, the protocol is believed to be *shortcut free*–audit responses from SSPs are guaranteed to embody end-to-end checks on data storage– under the assumption that collision resistant modification detection codes [48] exist. Second, the protocol is *externally verifiable* and *non-repudiable*—falsified SSP audit replies are quickly detected (with high probability) and deliberate falsifications can be proven to any third party[1].

4.1 Audit protocol

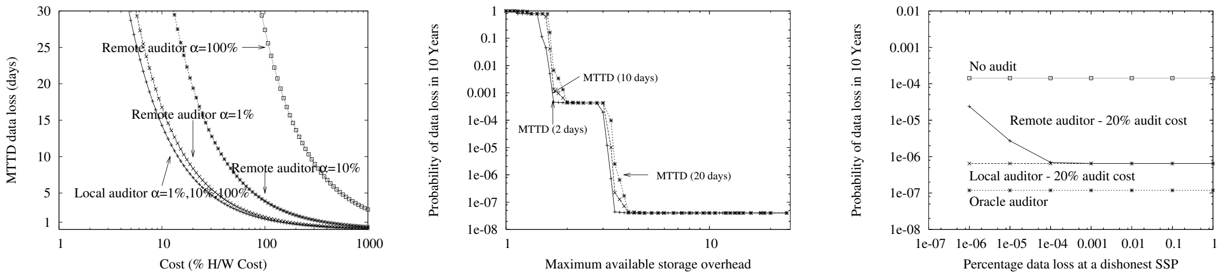
The audit protocol proceeds in three phases: (1) data storage, (2) routine audit, and (3) spot check. Note that the auditor may be co-located with or separate from the owner. For example, audit may be outsourced to an external auditor when data owners are offline for extended periods. To authorize SSPs to respond to auditor requests, the owner signs a certificate granting audit rights to the auditor's public key, and all requests from the auditor are authenticated against such a certificate (these authentication handshakes are omitted in the description below.) We describe the high level protocol here and detail it in Appendix B.

Data storage. When an object is stored at an SSP, the SSP signs and returns to the data owner a *receipt* that includes the object ID, cryptographic hash of the data, and storage expiration time. The data owner in turn verifies that the signed hash matches the data it sent and that the receipt is not malformed with an incorrect id or expiration time. If the data and hash fail to match, the owner retries sending the write message (data could have been corrupted in the transmission); repeated failures indicate a malfunctioning SSP and generate a notification to the data owner. As we detail in Section 5, SSPs do not provide a delete interface, so the expiration time indicates when the SSP will garbage collect the data. The data owner collects such valid receipts, encodes them, and spreads them across SSPs for durable storage.

Routine audit. The auditor sends to an SSP a list of object IDs and a random challenge. The SSP computes a cryptographic hash on both the challenge and the data. The SSP sends a signed message to the auditor that includes the object IDs, the current time, the challenge, and the hash computed on the challenge and the data ($H(challenge + data_{objId})$). The auditor buffers the challenge responses if the messages are well-formed, where a message is considered to be well-formed if none of the following conditions are true: the signature does not match the message, the response with an unacceptably stale timestamp, the response with the wrong challenge, or the response indicates error code (e.g., he SSP detected data is corrupt via internal checks or the data has expired). If the auditor does not receive any response from the SSP or if it receives a malformed message, the auditor notifies the data owner, and the data owner reconstructs the data via cached state or other SSPs and stores the lost fragment again. Of course, the owner may choose to switch SSPs before restoring the data and/or may extract penalties under their service level agreement (SLA) with the SSP, but such decisions are outside the scope of the protocol.

We conjecture that the audit response is shortcut free: an SSP must possess object's data to compute the correct hash. An honest SSP verifies the data integrity against the challenge-free hash stored at the creation time before sending a well-formed challenge response. If the integrity check fails (data is lost or corrupted) it sends the error code for lost data to the auditor. However, a *dishonest* SSP can choose to send a syntactically well-

[1] We assume that provably deliberate falsification can be punished via contractual or other out-of-band means [54], but details are outside the scope of this paper.



(a) (b) (c)

Fig. 6: (a) Time to detect SSP data loss via audit with varying amounts of resources dedicated to audit overhead assuming honest SSPs. (b) Durability with varying MTTD. (c) Impact on overall durability with a dishonest SSP. The audit cost model for hardware, storage, and network bandwidth are described in [46]

formed audit response with bogus hash value when the data is corrupted or lost. Note that the auditor just buffers well-formed messages and does not verify the integrity of the data objects covered by the audit in this phase. Yet, routine audits serve two key purposes. First, when performed against honest SSPs, they provide end-to-end guarantees about the integrity of the data objects covered by the audit. Second, they force dishonest SSPs to produce a signed, non-repudiable statement about the integrity of the data objects covered by the audit.

Spot check. In each round, after it receives audit responses in the routine audit phase, the auditor randomly selects α% of the objects to be spot checked. The auditor then retrieves each object's data (via the owner's cache, via the SSP, or via other SSPs) and verifies that the cryptographic hash of the challenge and data matches the challenge response sent by the SSP in the routine audit phase. If there is a mismatch, the auditor informs the data owner about the mismatch and provides the signed audit response sent by the SSP. The data owner then can create an externally-verifiable proof of misbehavior (POM) [46] against the SSP: the receipt, the audit response, and the object's data. In particular, the receipt is a signed statement with a hash of the data; the audit reply a signed claim to be storing the data and that a hash across a challenge and the data has a particular value; and the data allows anyone to verify that the receipt and audit reply refer to that data but that the challenge computation was incorrect. Note that SafeStore local server encrypts all data before storing it to SSPs, so this proof may be presented to third parties without leaking the plaintext object contents. Also, note that our protocol works with erasure coding as the auditor can reconstruct the data to be spot checked using redundant data stored at other SSPs.

4.2 Durability and cost

In this section we examine how the low-cost audit protocol limits the damage from faulty SSPs. The SafeStore protocol specifies that SSPs notify data owners immediately of any data loss that the SSP cannot internally recover so that the owner can restore the desired replication level using redundant data. Figures 4 and 5 illustrate the durability of our system when the SSPs follow the requirement and immediately report failures. As explained below, Figure 6-(a) and (b) show that SafeStore still provides excellent data durability with low audit cost, if a data owner is unlucky and selects a *passive* SSP that violates the immediate-notify requirement and waits for an audit of an object to report that it is missing. Figure 6-(c) shows that if a data owner is really unlucky and selects a *dishonest* SSP that first loses some of the owner's data and then lies when audited to try to conceal that fact, the owner's data is still very likely to emerge unscathed. We evaluate our audit protocol with 1TB of data stored redundantly across three SSPs with inter-SSP encoding of (3,1) (Appendix D has results for (3,2) encoding).

First, assume that SSPs are *passive* and wait for an audit to check data integrity. Because the protocol uses relatively cheap processing at the SSP to reduce data transfers across the wide area network, it is able to scan through the system's data relatively frequently without raising system costs too much. Figure 6-(a) plots the mean time to detect data loss (MTTD) at a *passive* SSP as a function of the cost of hardware resources (storage, network, and cpu) dedicated to auditing, expressed as a percentage of the cost of the system's total hardware resources as detailed in the caption. We also vary the fraction of objects that are spot checked in each audit round (α) for both the cases with local (co-located with the data owner) and remote (separated over WAN) auditors. We reach following conclusions: (1) As we increase the audit budget we can audit more frequently and the time to detect data loss falls rapidly. (2) audit costs with local and remote auditors is almost the same when α is less than 1%. (3) The audit cost with local auditor does not vary much with increasing α (as there is no additional network overhead in retrieving data from the local data owner) whereas the audit cost for the remote auditor increases with increasing α (due to additional network overhead in retrieving data over the WAN). (4) Overall, if a system dedicates 20% of resources to auditing, we can detect a lost data block within a week (with a local or a remote auditor with α = 1%).

Given this information, Figure 6-(b) shows the modest impact on overall data durability of increasing the time to detect and correct such failures when we assume that all SSPs are *passive* and SafeStore relies on auditing rather than immediate self reporting to trigger data recovery.

Now consider the possibility of an SSP trying to brazen its way through an audit of data it has lost using a made-up value purporting to be the hash of the challenge and data. The BAR model [30] argues for reasoning about systems spanning multiple administrative domains by assuming that most entities are rational and will act to maximize their utility and that a small number may be Byzantine and may act arbitrarily.he audit protocol encourages rational SSPs that lose data to respond to audits honestly. In particular, we prove in Appendix C that under reasonable assumptions about the penalty for an honest failure versus the penalty for generating a proof of misbehavior (POM), a rational SSP will maximize its utility [30] by faithfully executing the audit protocol as specified.

But suppose that through misconfiguration, malfunction, or malice, a node first loses data and then issues *dishonest* audit replies that claim that the node is storing a set of objects that it does not have. The spot check protocol ensures that if a node is missing even a small fraction of the objects, such cheating is quickly discovered with high probability. Furthermore, as that fraction increases, the time to detect falls rapidly. The intuition is simple: the probability of detecting a dishonest SSP in k audits is given by

$$p_k = 1-(1-p)^k$$

where p is the probability of detection in an audit, which is given by

$$p = \frac{\sum_{i=1}^{m}\binom{n}{i}\binom{N-n}{m-i}}{\binom{N}{m}}, (\text{if } n \geq m)$$

$$p = \frac{\sum_{i=1}^{n}\binom{m}{i}\binom{N-m}{n-i}}{\binom{N}{n}}, (\text{if } n < m)$$

where N is the total number of data blocks stored at an SSP, n is the number of blocks that are corrupted or lost and m is the number of blocks that are spot checked, α=(m/N) $\times$ 100.

WriteReceipt **write**(ID oid, byte data[], int64 size, int32 type, int64 expire);	
ReadReply **read**(ID oid, int64 size, int32 type)	
AttrReply **get_attr**(ID oid);	
TTLReceipt **extend_expire**(ID oid, int64 expire);	

Table 2: SSP storage interface

Figure 6-(c) shows the overall impact on durability if a node that has lost a fraction of objects maximizes the time to detect these failures by generating *dishonest* audit replies. We fix the audit budget at 20% and measure the durability of SafeStore with local auditor (with α at 100%) as well as remote auditor (with α at 1%). We also plot the durability with *oracle detector* which detects the data loss immediately and triggers recovery. Note that the *oracle detector* line shows worse durability than the lines in Figure 6-(b) because (b) shows durability for a randomly selected 10-year period while (c) shows durability for a 10-year period that begins when one SSP has already lost data. Without auditing (*no audit*), there is significant risk of data loss reducing durability by three 9's compared to *oracle detector*. Using our audit protocol with *remote auditor*, the figure shows that a cheating SSP can introduce a non-negligible probability of small-scale data loss because it takes multiple audit rounds to detect the loss as it spot checks only 1% of data blocks. But that the probability of data loss falls quickly and comes closer to *oracle detector* line (with in one 9 of durability) as the amount of data at risk rises. Finally, with a *local auditor*, data loss is detected in one audit round independent of data loss percentage at the dishonest SSPs as a local auditor can spot check all the data. In the presence of dishonest SSPs, our audit protocol improves durability of our system by two 9's over a system with no audit at an additional audit cost of just 20%. The overall durability of our system improves with increasing audit budget and approaches the *oracle detector* line as described in Appendix D.

5 SSFS

We implement SSFS, a file system that embodies the SafeStore architecture and protocol. In this section, we first describe the SSP interface and our SSFS SSP implementation. Then, we describe SSFS's local server.

5.1 SSP

As Figure 1 shows, for long-term data retention SSFS local servers store data redundantly across administratively autonomous SSPs using erasure coding or full replication. SafeStore SSPs provide a simple yet carefully defined object store interface to local servers as shown in Table 2.

Two aspects of this interface are important. First, it provides non-repudiable receipts for writes and expiration extensions in order to support our spot-check-based audit protocol. Second, it provides *temporal isolation* to limit the data owner's ability to change data that is currently stored [47]. In particular, the SafeStore SSP protocol (1) gives each object an absolute expiration time and (2) allows a data owner to extend but not reduce an object's lifetime.

The temporal isolation guarantee is as follows: if an SSP is storing a desired set of data at time t, an owner can ensure that the current version is accessible until any desired time in the future even if the local server suffers an arbitrary failure.

This interface supports what we expect to be a typical usage pattern in which an owner creates a ladder of backups at increasing granularity [62]. Suppose the owner wishes to maintain yearly backups for each year in the past 10 years, monthly backups for each month of the current year, weekly backups for the last four weeks, and daily backups for the last week. Using the local server's snapshot facility (see Section 5.2), on the last day of the year, the local server *writes* all current blocks that are not yet at the SSP with an expiration date 10-years into the future and also iterates across the most recent version of all remaining blocks and sends *extend_expire* requests with an expiration date 10-years into the future. Similarly, on the last day of each month, the local server writes all new blocks and extends the most recent version of all blocks; notice that blocks not modified during the current year may already have expiration times beyond the 1-year target, but these extensions will not reduce this time. Similarly, on the last day of each week, the local server writes new blocks and extends deadlines of the current version of blocks for a month. And every night, the local server writes new blocks and extends deadlines of the current version of all blocks for a week. Of course, SSPs ignore *extend_expire* requests that would shorten an object's expiration time.

SSP implementation. We have constructed a prototype SSFS SSP that supports all of the features described in this paper including the interface for servers and the interface for auditors. Internally, each SSP spreads data across a set nodes using erasure coding with a redundancy level specified for each data owner's account at account creation time.

For compatibility with legacy SSPs, we also implement a simplified SSP interface that allows data owners to store data to Amazon's S3 [1], which provides a simple non-versioned read/write/delete interface and which does not support our optimized audit protocol.

Issues. There are three outstanding issues in our current implementation. We believe all are manageable. First, the approach relies on prompt failure/intrusion detection: the shorter the period of time between when a fault mistakenly deletes/modifies an object and the owner realizes that she would prefer an older version, the more current backup that will be available. For simple failures (e.g., total disk failure), it will be easy for a data owner to quickly notice a problem. For more complex failures (e.g., malware that randomly modifies one bit in one file per day), detecting the problem is more difficult. We do not advance the state of the art in intrusion detection or fault detection, but we encourage data owners to make use of available tools [58, 65].

Second, in practice, it is likely that SSPs will provide some protocol for deleting data early. We assume that any such out-of-band early-delete mechanism is carefully designed to maximize resistance to erroneous deletion by the data owner. For concreteness, we assume that the payment stream for SSP services is well protected by the data owner and that our SSP will delete data 90 days after payment is stopped. So, a data owner can delete unwanted data by creating a new account, copying a subset of data from the old account to the new account, and then stopping payment on the old account. More sophisticated variations (e.g., using threshold-key cryptography to allow a quorum of independent administrators to sign off on a delete request) are possible.

Third, SSFS is vulnerable to resource consumption attacks: although an attacker who controls an owner's local server cannot reduce the integrity of data stored at SSPs, the attacker can send large amounts of long-lived garbage data and/or extend expirations farther than desired for large amounts of the owner's data stored at the SSP. We conjecture that SSPs would typically employ a quota system to bound resource consumption to within some budget along with an out-of-band early delete mechanism such as described in the previous paragraph to recover from any resulting denial of service attack.

5.2 Local Server

Clients interact with SSFS through a local server. The SSFS local server is a user level file system that exports the NFS 2.0 interface to its clients. The local server serves requests from local storage to improve the cost, performance, and availability of the system. Remote storage is used to store data durably to guard against local failures. The local server encrypts (using SHA1 and 1024 bit Rabin key signature) and encodes [57] (if data is not fully replicated) all data before sending it to remote SSPs, and it transparently fetches, decodes and decrypts data from remote storage if it is not present in the local cache. Our implementation thus supports policies that reduce local space demands by garbage collecting cold objects, but exploring such policies is future work; our prototype local server simply stores local copies of all objects.

All local server state except the encryption key and list of SSPs is soft state: given these items, the local server can recover the full filesystem. We assume both are stored out of band (e.g., the owner burns them to a CD at installation time and stores the CD in a safety deposit box). A more convenient (and thus more robust in terms of data durability) but lower-security alternative is to remember the list of SSPs and to encrypt the key with a password, erasure code it, and store the key fragments in well-known object IDs at the SSPs.

Snapshots: In addition to the standard NFS calls, the SSFS local server provides a snapshot interface [16] that supports file versioning for achieving temporal isolation to tolerate client or administrator failures. A snapshot stores a copy in the local cache and also redundantly stores encrypted, erasure-coded data across multiple SSPs using the remote storage interface.

Local storage is structured carefully to reduce storage and performance overheads for maintaining multiple versions of files. SSFS uses block-level versioning [16, 55] to reduce storage overhead by storing only modified blocks in the older versions when a file is modified. For each old version, SSFS maintains a *block mask* and *size* in a meta-data file for the older version. Then, reads of the current version see no overhead, and reads of the older version are satisfied by starting with the old version and then fetching data blocks not present in the old version from later versions by sequentially checking the later versions until the block is found [55]. And as an obvious extension for the common case of files modified by appends: on an append, SSFS needs only to store the old size (and not the block mask) as all blocks are stored in later versions.

Other optimizations: SSFS uses a fast recovery optimization to recover quickly from remote storage when local data is lost due to local server failures (disk crashes, fire, etc.) The SSFS local server recovers quickly by coming online as soon as all metadata information (directories, inodes, and old-version information) is recovered and then fetching file data to fill the local cache in the background. If a missing block is requested before it is recovered, it is fetched immediately on demand from the SSPs. Additionally, local storage acts as a write-back cache where updates are propagated to remote SSPs asynchronously so that client performance is not affected by updates to remote storage.

6 Evaluation

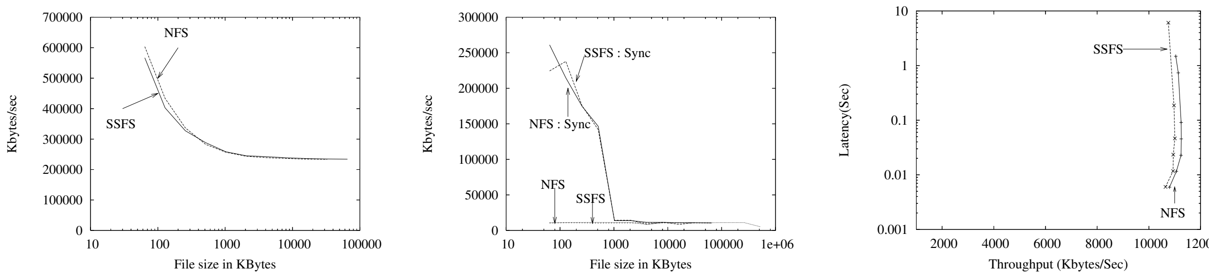
To evaluate the practicality of the SafeStore architecture, we evaluate our SSFS prototype via microbenchmarks selected to stress test three aspects of the design. First, we examine performance overheads, then we look at storage space overheads, and finally we evaluate recovery performance.

In our base setup, client, local server, and remote SSP servers run on different machines that are connected by a 100 Mbit isolated network. For several experiments we modify the network to synthetically model WAN behavior. All of our machines use 933MHZ Intel Pentium III processors with 256 MB RAM and run Linux version 2.4.7. We use (3,2) erasure coding or full replication ((3,1) encoding) to redundantly store backup data across SSPs.

6.1 Performance

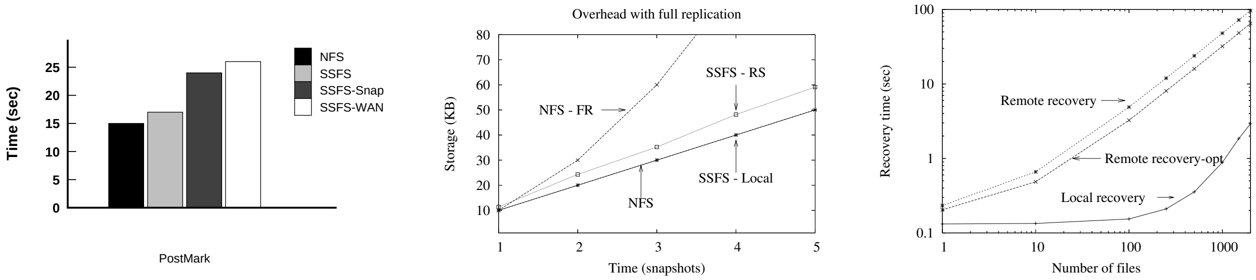
Figure 7 compares the performance of SSFS and a standard NFS server using the IOZONE [13] microbenchmark. In this experiment, we measure the overhead of SSFS's bookkeeping to maintain version information, but we do not take filesystem snapshots and hence no data is sent to the remote SSPs. Figure 7(a),(b), and (c) illustrates throughput for reads, throughput for synchronous and asynchronous writes, and throughput versus latency for SSFS and stand alone NFS. In all cases, SSFS's throughput is within 12% of NFS.

Figure 8(a) examines the cost of snapshots. Note SSFS sends snapshots to SSPs asynchronously, but we have not lowered the priority of these background transfers, so snapshot transfers can interfere with demand requests. To evaluate this effect, we add snapshots to the Postmark [19] benchmark, which models email/e-commerce workloads. The benchmark initially creates a pool of files and then performs a specified number of transactions consisting of creating, deleting, reading, or appending a file. We set file sizes to be between 100B and 100KB and run 50000 transactions. To maximize the stress on SSFS, we set the Postmark parameters to maximize the fraction of append and create operations. Then, we modify the benchmark to take frequent snapshots: we tell the server to create a new snapshot after every 500 transactions. As shown in the Figure 8(a), when no snapshots are taken SSFS takes 13% more time than NFS due to overhead involved in maintaining multiple versions. Turning on frequent snapshots increases the response time of SSFS (SSFS-snap in Figure 8(a)) by 40% due to additional overhead due to signing and transmitting updates to SSPs. Finally, we vary network latencies to SSPs to study the impact of WAN latencies on performance when SSPs are geographically distributed



(a) (b) (c)

Fig. 7: IOZONE : (a) Read (b) Write (c) Latency versus Throughput



(a) (b) (c)

Fig. 8: (a) Postmark: End-to-end performance (b) Storage overhead (c) Recovery

over the Internet by introducing artificial delay (of 40 ms) at the SSP server. As shown in the Figure 8(a), SSFS-WAN response time increases by less than an additional 5%.

6.2 Storage overhead

Here, we evaluate the effectiveness of SSFS's mechanisms for limiting replication overhead. SSFS minimizes storage overheads by using a versioning system that stores the difference between versions of a file rather than complete copies [55]. We compare the storage overhead of SSFS's versioning file system and compare it with NFS storage that just keeps a copy of the latest version and also a naive versioning NFS file system (NFS-FR) that makes a complete copy of the file before generating a new version. Figure 8(b) plots the storage consumed by local storage (SSFS-LS) and storage at one remote server (SSFS-RS) when we use a (3,1) encoding. To expose the overheads of the versioning system, the microbenchmark is simple: we append 10KB to a file after every file system snapshot. SSFS's local storage takes a negligible amount of additional space compared to non-versioned NFS storage. Remote storage pays a somewhat higher overhead due to duplicate data storage when appends do not fall on block boundaries and due to additional metadata (integrity hashes, the signed write request, expiry time of the file, etc.)

We also ran an experiment with the (3,2) encoding at remote servers using Postmark benchmark with varying snapshot frequencies and observed similar results. We omit these graphs for brevity. The above experiments examine the case when the old and new versions of data have much in common and test whether SSFS can exploit such situations with low overhead. There is, of course, no free lunch: if there is little in common between a user's current data and old data, the system must store both. Like SafeStore, Glacier uses a expire-then-garbage collect approach to avoid inadvertent file deletion, and their experience over several months of operation is that the space overheads are reasonable [41]. We plan to confirm these results in a SafeStore context by evaluating space overhead using the long-duration Harvard traces [38].

6.3 Recovery

We now evaluate SSFS recovery time and compare performance with and without SSFS's fast recovery optimization that allows the local server to resume operation as soon as it has recovered file system metadata and to recover the rest of the system's data in the background.

We also plot recovery time of SSFS from local storage due to reboots of the local server. Figure 8(c) plots recovery time as the number of 1KB files in the system varies when the data is recovered from remote SSPs. We see that local recovery is faster than the other two as it recovers from the local disk and it outperforms the other two by more than an order of magnitude for moderate number of files in the system. We also observe that remote recovery with optimization outperforms remote recovery without optimization by about 50% even with as few as 10 files. Note that recovery time is high even with the optimization as SSFS recovers all the metadata (which involves reading from remote SSPs, verifying the metadata integrity, decoding data from redundant fragments, and finally decrypting the metadata) before it starts serving the client requests. As part of our future work, we intend to reduce the recovery time significantly by bringing the system up immediately while the metadata is fetched in the background like the existing optimization for data.

7 Related work

Several recent studies [31, 60] have identified the challenges involved in building durable storage system for multi-year timescales.

Flat erasure coding across nodes [33, 36, 41, 69] does not require detailed predictions of which sets of nodes are likely to suffer correlated failures because it tolerates any combinations of failures up to a maximum number of nodes. However, flat encoding does not exploit the opportunity to reduce replication costs when the system can be structured to make some failure combinations more likely than others. An alternative approach is to use *full replication* across sites that are not expected to fail together [44, 47], but this can be expensive.

SafeStore is architected to increase the likelihood that failures will be restricted to specific groups of nodes, and it efficiently deploys storage within and across SSPs to address such failures. Myriad [34] also argues for a 2-level (cross-site, within-site) coding strategy, but SafeStore's architecture departs from Myriad in keeping SSPs at arms-length from data owners by carefully restricting the SSP interface and by including provisions for efficient end-to-end auditing of black-box SSPs.

SafeStore is most similar in spirit to OceanStore [43] in that we erasure code indelible, versioned data across independent SSPs. But in pursuit of a more aggressive "nomadic data" vision, OceanStore augments this approach with a sophisticated overlay-based infrastructure for replication of location-independent objects that may be accessed concurrently from various locations in the network [56]. We gain considerable simplicity by using a local soft-state server through which all user requests pass and by focusing on storing data on a relatively small set of specific, relatively conventional SSPs. We also gain assurance in the workings of our SSPs through our audit protocol.

Versioning file systems [16, 51, 59, 62, 67] provide temporal isolation to tolerate client failures by keeping multiple versions of files. We make use of this technique but couple it with efficient, isolated, audited storage to address a broader threat model.

We argue that highly durable storage systems should audit data periodically to ensure data integrity and to limit worst-case MTTR. Zero-knowledge-based audit mechanisms [39, 48] are either network intensive or CPU intensive as their main purpose is to audit data without leaking any information about the data. SafeStore avoids the need for such expensive approaches by encrypting data before storing it. We are then able to offload audit duties to SSPs and probabilistically spot check their results. LOCKSS [47] and Samsara [37] audit data in P2P storage systems but assume that peers store full replicas so that they can easily verify if peers store identical data. SafeStore supports erasure coding to reduce costs, so our audit mechanism does not require SSPs to have fully replicated copies of data.

8 Conclusion

Achieving robust data storage on the scale of decades forces us to reexamine storage architectures: a broad range of threats that could be neglected over shorter timescales must now be considered. SafeStore aggressively applies the principle of *fault isolation* along administrative, physical, and temporal dimensions. Analysis indicates that SafeStore can provide highly robust storage and evaluation of an NFS prototype suggests that the approach is practical.

9 Acknowledgements

This work was supported in part by NSF grants CNS-0411026, CNS-0430510, and CNS-0509338 and by the Center for Information Assurance and Security at the University of Texas at Austin.

References

[1] Amazon S3 Storage Service. http://aws.amazon.com/s3.
[2] Apple Backup. http://www.apple.com.
[3] Concerns raised on tape backup methods. http://searchsecurity.techtarget.com.
[4] Copan Systems. http://www.copansys.com/.
[5] Data loss statistics. http://www.hp.com/sbso/serverstorage/protect.html.
[6] Data loss statistics. http://www.adrdatarecovery.com/content/adr_loss_stat.html.
[7] Fire destroys research center. http://news.bbc.co.uk/1/hi/england/hampshire/4390048.stm.
[8] Health Insurance Portability and Accountability Act (HIPAA). 104th Congress, United States of America Public Law 104-191.

[9] Hotmail incinerates customer files. http://news.com.com, June 3rd, 2004.

[10] "How much information ?". http://www.sims.berkeley.edu/projects/how-much-info/.

[11] Hurricane Katrina. http://en.wikipedia.org.

[12] Industry data retention regulations. http://www.veritas.com/van/articles/4435.jsp.

[13] IOZONE micro-benchmarks. http://www.iozone.org.

[14] Lost Gmail Emails and the Future of Web Apps. http://it.slashdot.org, Dec 29, 2006.

[15] NetMass Systems. http://www.netmass.com.

[16] Network Appliance. http://www.netapp.com.

[17] Network bandwidth cost. http://www.broadbandbuyer.com/formbusiness.htm.

[18] OS vulnerabilities. http://www.cert.com/stats.

[19] Postmark macro-benchmark. http://www.netapp.com/tech_library/postmark.html.

[20] Ransomware. http://www.networkworld.com/buzz/2005/092605-ransom.html.

[21] Remote Data Backups. http://www.remotedatabackup.com.

[22] Sarbanes-Oxley Act of 2002. 107th Congress, United States of America Public Law 107-204.

[23] Spike in Laptop Thefts Stirs Jitters Over Data. Washington Post, June 22, 2006.

[24] SSPs: RIP. Byte and Switch, 2002.

[25] Tape Replacement Realities. http://www.enterprisestrategygroup.com/ESGPublications.

[26] The Wayback Machine. http://www.archive.org/web/hardware.php.

[27] US secret service report on insider attacks. http://www.sei.cmu.edu/about/press/insider-2005.html.

[28] Victims of lost files out of luck. http://news.com.com, April 22, 2002.

[29] "data backup no big deal to many, until...". http://money.cnn.com, June 2006.

[30] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. of SOSP '05*, pages 45–58, Oct. 2005.

[31] M. Baker, M. Shah, D.S.Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *EuroSys*, 2006.

[32] L. Bassham and W. Polk. Threat assessment of malicious code and human threats. Technical report, National Institute of Standards and Technology Computer Security Division, 1994. http://csrc.nist.gov/nistir/threats/.

[33] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. Total Recall: System support for automated availability management. In *Proceedings of 1st NSDI*, CA, 2004.

[34] F. Chang, M. Ji, S. T. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proceeedings of FAST*, 2002.

[35] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID : High-performance,reliable secondary storage. *ACM Comp. Surveys*, 26(2):145–185, June 1994.

[36] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM Conference on Digital Libraries*, San Fransisco, CA, Aug 1999.

[37] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. of SOSP03*.

[38] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *FAST03*, Mar. 2003.

[39] P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In *Financial Cryptography (FC 2002)*, volume 2357 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2003.

[40] J. Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–418, Oct. 1990.

[41] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Higly durable, decentralized storage despite massive correlated failures. In *Proceedings of 2nd NSDI*, CA, March 2004.

[42] R. Hassan, W. Yurcik, and S. Myagmar. The evolution of storage service providers. In *StorageSS'05*, VA,USA, 2005.

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

[44] F. Junquiera, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. Surviving internet catastrophes. In *Proceedings of the Usenix Annual Technical Conference*, April 2005.

[45] K. Keeton and E. Anderson. A backup appliance composed of high-capacity disk drives. In *HP Laboratories SSP Technical Memo HPL-SSP-2001-3*, April 2001.

[46] R. Kotla, L. Alvisi, and M. Dahlin. Safestore: A durable and practical storage system. Technical report, University of Texas at Austin, 2007. UT-CS-TR-07-20.

[47] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, M. Baker, and Y. Muliadi. Lockss: A peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, Feb. 2005.

[48] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Hndbook of Applied Cryptography*. CRC Press, 2001.

[49] D. Openheimer, A. Ganapathi, and D. Patterson. Why do internet systems fail, and what can be done about it. In *Proceedings of 4th USITS*, Seattle,WA, March 2003.

[50] D. Patterson. A conversation with jim gray. *ACM Queue*, pages vol. 1, no. 4, June 2003.

[51] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Trans. on Storage*, 1(2):190–212, May. 2005.

[52] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proceeedings of FAST*, 2007.

[53] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. G. A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *Proc. of SOSP '05*, 2005.

[54] H. E. Ramadan. Abort, retry, litigate: Dependable systems and contract law. In *Proceedings of HotDep '06*, 2006.

[55] K. M. Reddy, C. P. Wright, A. Hammer, and E. Zadok. A Versatile and user-oriented versioning file system. In *FAST*, 2004.

[56] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST03*, Mar. 2003.

[57] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Comp. Comm. Review*, 27(2), 1997.

[58] M. Roesch. Snort–lightweight intrusion detection for networks. In *Proc LISA*, 1999.

[59] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[60] D. S. H. Rosenthal, T. S. Robertson, T. Lipkis, V. Reich, and S. Morabito. Requirements for digital preservation systems: A bottom-up approach. *D-Lib Magazine*, 11(11), Nov. 2005.

[61] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM TOCS*, Nov. 1984.

[62] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of 17th ACM Symp. on Operating Systems Principles*, December 1999.

[63] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceeedings of FAST*, 2007.

[64] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, and S. Ng.

Disk scrubbing in large archival storage systems. In *Proc. MASCOTS*, Oct. 2004.

[65] Seagate. Get S.M.A.R.T for reliability. Technical Report TP-67D, Seagate, 1999.

[66] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of 6th OSDI*, 2004.

[67] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in a comprehensive versioning file system. In *Proc. of FAST 2003*.

[68] H. Weatherspoon and J. Kubiatowicz. Erasure Coding versus replication: A quantitative comparison. In *Proceedings of IPTPS*, Cambridge,MA, March 2002.

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

[70] Q. Xin, T. Schwarz, and E. Miller. Disk infant mortality in large storage systems. In *Proc of MASCOTS '05*, 2005.

[71] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of 6th OSDI*, December 2004.

A Durability analysis

Here, we describe the analytical models for analyzing durability of hierarchical encoding, flat erasure coding, and full replication. Consider a system with n nodes spread across k groups (SSPs) with n_1, n_2,..n_k nodes respectively present in groups 1,2,...k. All nodes in a group n_i fail in a correlated fashion with probability p_c due to a correlated failure events (fire, administrator failure) at an SSP. Also, a node fails independently with a failure probability p_u due to uncorrelated failure events (disk failures) at an SSP. In order to analyze durability over some time duration, we first evaluate the durability of data in an epoch and then aggregate it over multiple epochs spanning the duration as in [68]. Given this failure model, durability of hierarchical encoding, flat erasure coding, and full replication is described below

Overhead : Storage overhead of the system is

$$= 1/l \times (n_0/m_0 + n_1/m_1 + ... + n_{k-1}/l_{k-1})$$
$$= 1/l \times (r_0 + r_1 + ... + r_{k-1})$$

where r_0, r_1, ...r_{k-1} are the intra-SSP redundancies used by SSPs 0,1,...k-1. The inter-SSP redundancy is k/l when (k,l) encoding is used to store data redundantly across SSPs.

Hierarchical encoding: Hierarchical encoding of data with (k,l) erasure coding across SSPs and with (n_i, m_i) encoding within an SSP group i is given by

$$\text{Random variable} : X_i$$
$$X_i = 0, \text{if} < m_i \text{ nodes are up in SSP i}$$
$$= 1, \text{if} \geq m_i \text{ nodes are up in SSP i}$$

$$Pr(X_i = 1) = (1 - p_c) \times \sum_{j=m_i}^{n_i} \binom{n_i}{j} (1 - p_u)^j p_u^{n_i - j},$$

$$\text{Durability of data in an epoch, } D_e = Pr(\sum_{i=1}^{k} X_i \geq l)$$

Durability: Overall durability of data (using any replication mechanism) for a time duration T is given by

Durability of data in time duration T, D

= 1-prob(data loss in time t $\leq$ T)

= ${D_e}^{T/e}$

where e is the epoch length and D_e is the durability of a given replication mechanism in an epoch.

We set epoch length to MTTR min($MTTR_u$,$MTTR_c$), where $MTTR_u$ and $MTTR_c$ are mean time to receoveries from uncorrelated node failure with in SSP and correlated SSP failures. We assume that failures in an epoch are not repaired before the end of epoch in computing D_e. The failures are assumed to be repaired before the start of next epoch (as we assume each epoch instance as a fresh Bernoulli trial while computing overall durability D as shown above). Given this epoch length, we can compute p_u and p_c from MTTDL [31] due to ucorrelated *node failure* ($MTTDL_u$) and correlated *SSP failure* ($MTTDL_c$) events as

$$p_u = MTTR_u / MTTDL_u$$
$$p_c = MTTR_c / MTTDL_c$$

Observation 1: The overall durability does not improve much by additional intra-SSP redundancies beyond a certain minimum value when $p_u << 1 - p_u$. For most practical values of $MTTDL_u$ (due to *node failure* of more than few years) and $MTTR_u$, $p_u << 1 - p_u$. For example, $p_u/(1 - p_u) < 0.0002$

for MTTDL due to node failure is 5 years and MTTR of 1 day. It is explained as follows

$$Pr(X_i = 1) = (1-p_c) \times \sum_{j=m_i}^{n_i} \binom{n_i}{j} (1-p_u)^j p_u^{n_i-j},$$

$$= (1-p_c) \times (1-p_u)^{n_i} \times (1 + \sum_{j=1}^{n_i-m_i} \binom{n_i}{j} (p_u/(1-p_u))^j),$$

$$\approx (1-p_c) \times (1-p_u)^{n_i} \times (1 + \sum_{j=1}^{\alpha} \binom{n_i}{j} (p_u/(1-p_u))^j),$$

$$= (1-p_c) \times \sum_{j=\alpha}^{n_i} \binom{n_i}{j} (1-p_u)^j p_u^{n_i-j},$$

where α depends on n_i and $p_u/(1-p_u)$, such that $(\alpha+1)$ ¿¿ $(n_i-1) \times p_u/(1-p_u))$. For example, $\alpha \approx n_i - 1$, for n_i (tens of nodes), $MTTDL_u$ and $MTTR_u$ values of 5 years and 1 day. This implies that overall durability does not improve much beyond intra-SSP redundancy of $n_i/(n_i-1)$.

Mean time to data loss (MTTDL): Durability of data in terms of MTTDL is given by

$$\text{MTTDL} = \sum_{i=0}^{\infty} i{D_e}^i (1-D_e) = D_e/(1-D_e)$$

where D_e is the durability of a given replication mechanism in an epoch.

B Audit protocol

1. Data storage:
   O : Data owner SSP,SSP' : Storage service providers
   $O \rightarrow SSP : \{objId, H(data_{objId}), expire\}_O, data_{objId}$
   $SSP \rightarrow O : \{objId, H(data_{objId}), expire\}_{SSP}$
   $O \rightarrow SSP' : SSP_{id}, \{objId, H(data_{objId}), expire\}_{SSP}$

2. Routine audit:
   $A \rightarrow SSP : chal, listOfObjects$
   $SSP \rightarrow A : \{objId, chal, time, H(chal + data_{objId})\}_{SSP}$

3. Spot check:
   $A \rightarrow O|SSP|SSP' : list2OfObjects$
   $O|SSP|SSP' \rightarrow A : data_{objId}$

$$\begin{aligned} POM = \quad & receipt \text{ and } auditReply \text{ are well-formed} \\ & \text{and signed by } SSP \\ & objId = receipt_{objId} = auditReply_{objId} \\ & \wedge receipt_{expires} > auditReply_{time} \\ & \wedge receipt_{H(objId)} = auditReply_{H(objId)} \\ & \wedge chal = auditReply_{chal} \\ & \wedge H(chal + data) \neq auditReply_{H(chal+data)} \end{aligned}$$

C Audit analysis

Here we show that a rational [30] SSP (1) attempts to store data reliably and (2) responds to audit requests honestly assuming an SLA that specifies appropriate penalties relative to the underlying cost of storing data.

Definitions:

b_{serve}	Benefit for storing and serving object until it expires
c_{store}	Cost to store and serve object until it expires (including cost of serving audit requests)
p_{audit}	Probability that object will be audited before it expires
p_{spot}	Probability that an audit reply will be spot-checked
$penalty_h$	Penalty for *honest failure* of audit (see Section 4
$penalty_d$	Penalty for *dishonest failure* of audit

- $b_{serve} > c_{store} \wedge c_{store} < min(p_{audit} penalty_h, p_{audit} p_{spot} penalty_d) \implies$ A rational SSP attempts to store an object until it expires.
- $penalty_h < p_{spot} penalty_d \implies$ A rational SSP that does not have the data needed to reply to an audit request replies with an *honest failure* rather than with a audit reply that could be used to generate a proof of misbehavior.

Example. These requirements are met by a system with $c <$ \$1 (reasonable if all objects are broken into 1GB or smaller pieces and stored with expiration times of less than a year), $b = 2c$, $p_{audit} = 90\%$, $p_{spot} = 1\%$, $penalty_h =$ \$5, and $penalty_d =$ \$1000.

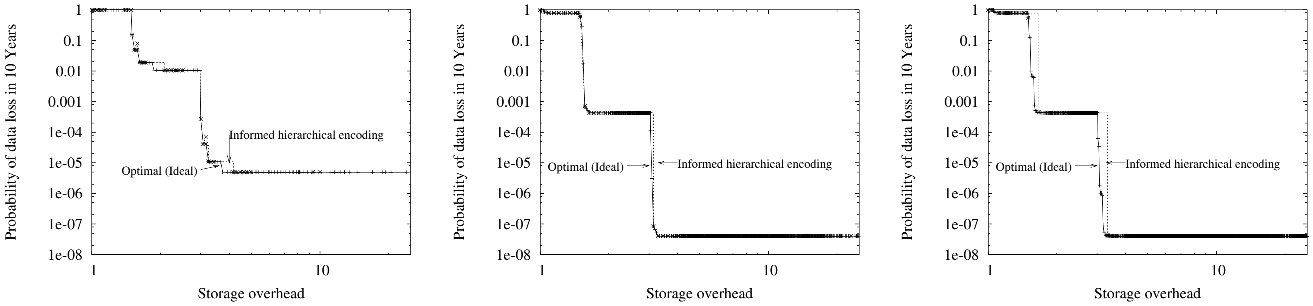
D Additional experiments

D.1 Informed hierarchical encoding

Here, we run additional experiments to study the sensitivity of our results to MTTDL and MTTR of *correlated failures* and total number of nodes in the syste. As shown in Figure 9, the conclusions made in section 3 continues to hold when MTTDL due to *correlated failures* is changed to 10 years from 100 years while MTTR is changes from 10 days to 5 days as *informed hierarchical encoding* continues to follow the optimal. Figure 9(b) shows that our conclusions are true even when we increase the number of nodes and also when they are non-uniformly distributed.

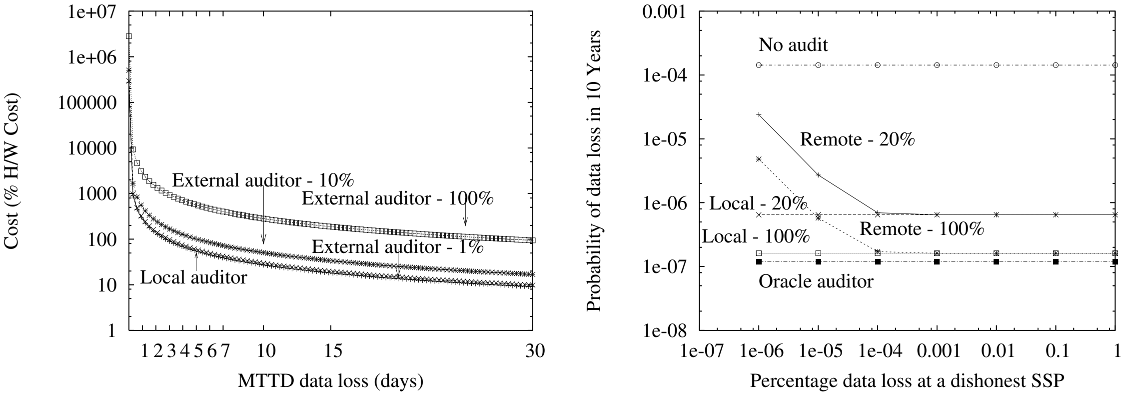
D.2 Audit

Here, we run additional experiments to evaluate our audit protocol as described in section 4. Figure 10(a) plots mean time to detect data loss (MTTD) at a *passive* SSP when (3,2) encoding is used to store 1TB of data redundantly across 3 SSPs. MTTD falls rapidly with increasing audit budget similar to the system that uses (3,1) as shown in the Figure 6(a) of section 4. However, for a fixed MTTD, (3,2) encoding incurs higher audit cost per byte stored compared to (3,1) because of increased overhead due to reduced block size from 4KB (with (3,1)) to 2KB (with (3,2)). Figure 10 illustrates the overall impact on durability in the presence of *dishonest* SSPs with varying audit budgets. With 20% audit budget, we outperform a system with no audit by two 9's in the presence of *dishonest* SSPs. As we increase our system's audit budget from 20% to 100%, durability of our system approaches that of a system with *oracle detector*.



(a) (b) (c)

Fig. 9: Informed hierarchical encoding (a) With MTTDL of *correlated failures* set to 10 years with MTTR of 5 days, (b) With 69 total nodes distributed uniformly across 3 SSPs, (c) With 69 nodes distributed non-uniformly across 3 SSPs as (10,20,39)



(a) (b)

Fig. 10: Audit (a) Time to detect SSP data loss via audit with varying amounts of resources dedicated to audit overhead assuming honest SSPs with (3,2) inter-SSP redundancy. (b) Impact on overall durability with a dishonest SSP with varying audit costs (20% and 100%)