Smoke and Mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance

Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, †Mahesh Balakrishnan, and Ken Birman
Cornell University, Computer Science Department, Ithaca, NY 14853
{hweather,lakshmi,tudorm,ken}@cs.cornell.edu

†*Microsoft Research, Silicon Valley*
maheshba@microsoft.com

Abstract

The Smoke and Mirrors File System (SMFS) mirrors files at geographically remote datacenter locations with negligible impact on file system performance at the primary site, and minimal degradation as a function of link latency. It accomplishes this goal using wide-area links that run at extremely high speeds, but have long round-trip-time latencies—a combination of properties that poses problems for traditional mirroring solutions. In addition to its raw speed, SMFS maintains good synchronization: should the primary site become completely unavailable, the system minimizes loss of work, even for applications that simultaneously update groups of files. We present the SMFS design, then evaluate the system on Emulab and the Cornell National Lambda Rail (NLR) Ring testbed. Intended applications include wide-area file sharing and remote backup for disaster recovery.

1 Introduction

Securing data from large-scale disasters is important, especially for critical enterprises such as major banks, brokerages, and other service providers. Data loss can be catastrophic for any company — Gartner estimates that 40% of enterprises that experience a disaster (e.g. loss of a site) go out of business within five years [41]. Data loss failure in a large bank can have much greater consequences with potentially global implications.

Accordingly, many organizations are looking at dedicated high-speed optical links as a disaster tolerance option: they hope to continuously mirror vital data at remote locations, ensuring safety from geographically localized failures such as those caused by natural disasters or other calamities. However, taking advantage of this new capability in the wide-area has been a challenge; existing mirroring solutions are highly latency sensitive [19]. As a result, many critical enterprises operate at risk of catastrophic data loss [22].

The central trade-off involves balancing safety against performance. So-called synchronous mirroring solutions [6, 12] block applications until data is safely mirrored at the remote location: the primary site waits for an acknowledgment from the remote site before allowing the application to continue executing. These are very safe, but extremely sensitive to link latency. Semi-synchronous mirroring solutions [12, 42] allow the application to continue executing once data has been written to a local disk; the updates are transmitted as soon as possible, but data can still be lost if disaster strikes. The end of the spectrum is fully asynchronous: not only does the application resume as soon as the data is written locally, but updates are also batched and may be transmitted periodically, for instance every thirty minutes [6, 12, 19, 31]. These solutions perform best, but have the weakest safety guarantees.

Today, most enterprises primarily use asynchronous or semi-synchronous remote mirroring solutions over the wide-area, despite the significant risks posed by such a stance. Their applications simply cannot tolerate the performance degradation of synchronous solutions [22]. The US Treasury Department and the Finance Sector Technology Consortium have identified the creation of new options as a top priority for the community [30].

In this paper, we explore a new mirroring option called *network-sync*, which potentially offers stronger guarantees on data reliability than semi-synchronous and asynchronous solutions while retaining their performance. It is designed around two principles. First, it proactively adds redundancy at the network level to transmitted data. Second, it exposes the level of in-network redundancy added for any sent data via feedback notifications. Proactive redundancy allows for reliable transmission with latency and jitter independent of the length of the link, a property critical for long-distance mirroring. Feedback makes it possible for a file system (or other applications) to respond to clients as soon as enough recovery data has been transmitted to ensure that the desired safety level has been reached. Figure 1 illustrates this idea.

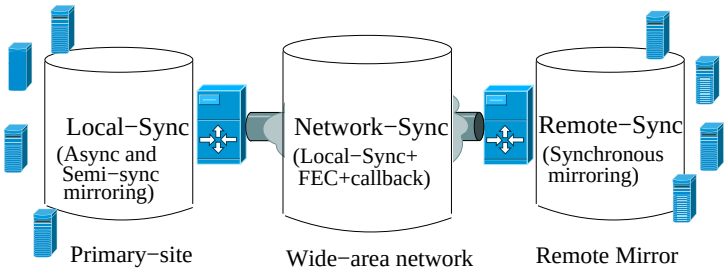

Figure 1: Remote Mirroring Options. (1) Synchronous mirroring provides a *remote-sync* guarantee: data is not lost in the event of disaster, but performance is extremely sensitive to the distance between sites. (2) Asynchronous and semi-synchronous mirroring give a *local-sync* guarantee: performance is independent of distance between mirrors, but can suffer significant data loss when disaster strikes. (3) A new *network-sync* mirroring option with performance similar to local-sync protocols, but with improved reliability.

Of course, data can still be lost; network-sync is not as safe as a synchronous solution. If the primary site fails and the wide-area network simultaneously partitions, data will still be lost. Such scenarios are uncommon, however. Network-sync offers the developer a valuable new option for trading data reliability against performance.

Although this paper focuses on the Smoke and Mirrors File System (SMFS), we believe that many kinds of applications could benefit from a network-sync option. These include other kinds of storage systems where remote mirroring is performed by a disk array (e.g. [12]), a storage area network (e.g. [19]), or a more traditional file server (e.g. [31]). Network-sync might also be valuable in transactional databases that stream update logs from a primary site to a backup, or to other kinds of fault-tolerant services.

Beyond its use of the network-sync option, SMFS has a second interesting property. Many applications update files in groups, and in such cases, if even one of the files in a group is out of date, the whole group may be useless (Seneca [19] calls this atomic, in-order asynchronous batched commits; SnapMirror [31] offers a similar capability). SMFS addresses the need in two ways. First, if an application updates multiple files in a short period of time, the updates will reach the remote site with minimal temporal skew. Second, SMFS maintains group-mirroring consistency, in which files in the same file system can be updated as a group in a single operation where the group of updates will all be reflected by the remote mirror site atomically, either all or none.

In summary, our paper makes the following contributions:

- We propose a new remote mirroring option called network-sync in which error-correction packets are proactively transmitted, and link-state is exposed through a callback interface.
- We describe the implementation and evaluation of SMFS, a new mirroring file system that supports both capabilities, using an emulated wide-area network (Emulab [40]) and the Cornell National Lambda Rail (NLR) Ring testbed [1]. This evaluation shows that SMFS:
  - Can be tuned to lose little or no data in the event of a rolling disaster.
  - Supports high update throughput, masking wide-area latency between the primary site and the mirror.
  - Minimizes jitter when files are updated in short periods of time.
- We show that SMFS has good group-update performance and suggest that this represents a benefit to using a log-structured file architecture in remote mirroring.

The rest of this paper is structured as follows. We discuss our fault model in Section 2. In Section 3, we describe the network-sync option. We describe the SMFS protocols that interact with the network-sync option in Section 4. In Section 5, we evaluate the design and implementation. Finally, Section 6 describes related work and Section 7 concludes.

2 What's the Worst that Could Happen?

We argue that our work responds to a serious imperative confronted by the financial community (as well as by other critical infrastructure providers). As noted above, today many enterprises opt to use asynchronous or semi-synchronous remote mirroring solutions despite the risks they pose, because synchronous solutions are perceived as prohibitively expensive in terms of performance [22]. In effect, these enterprises have concluded that there simply is no way to maintain a backup at geographically re-

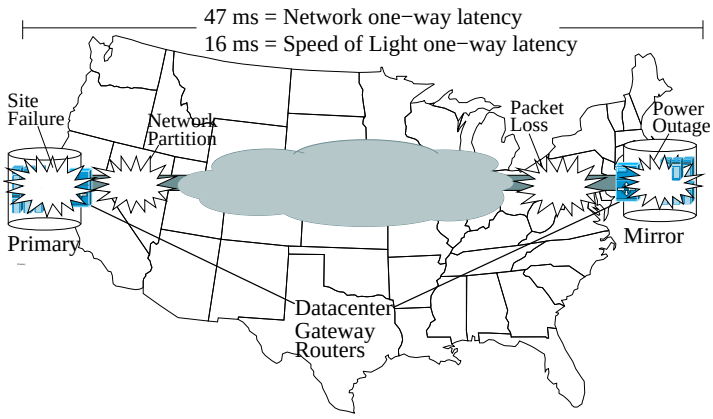

Figure 2: Example Failure Events. A single failure event may not result in loss of data. However, multiple nearly-simultaneous failure events (i.e. rolling disaster) may result in data loss for asynchronous and semi-synchronous remote mirroring.

mote distances at the update rates seen within their datacenters. Faced with this apparent impossibility, they literally risk disaster.

It is not feasible to simply legislate a solution, because today's technical options are inadequate. Financial systems are under huge competitive pressure to support enormous transaction rates, and as the clearing time for transactions continues to diminish towards immediate settlement, the amounts of money at risk from even a small loss of data will continue to rise [20]. Asking a bank to operate in slow-motion so as to continuously and synchronously maintain a remote mirrored backup is just not practical: the institution would fail for reasons of non-competitiveness.

Our work cannot completely eliminate this problem: for the largest transactions, synchronous mirroring (or some other means of guaranteeing that data will survive any possible outage) will remain necessary. Nonetheless, we believe that there may be a very large class of applications with intermediary data stability needs. If we can reduce the window of vulnerability significantly, our hypothesis is that even in a true disaster that takes the primary site offline and simultaneously disrupts the network, the challenges of restarting using the backup will be reduced. Institutions betting on network-sync would still be making a bet, but we believe the bet is a much less extreme one, and much easier to justify.

Failure Model and Assumptions: We assume that failures can occur at any level — including storage devices, storage area network, network links, switches, hubs, wide-area network, and/or an entire site. Further, we assume that they can fail simultaneously or even in sequence: a rolling disaster. However, we assume that the storage system at each site is capable of tolerating and recovering from all but the most extreme local failures. Also, sites may have redundant network paths connecting them. This allows us to focus on the tolerance of failures that disable an entire site, and on combinations of failures such as the loss of both an entire site and the network connecting it to the backup (what we call a rolling disaster). Figure 2 illustrates some points of failure.

With respect to wide-area optical links, we assume that even though industry standards essentially preclude data loss on the links themselves, wide-area connections include layers of electronics: routers, gateways, firewalls, etc. These components can and do drop packets, and at very high data rates, so can the operating system on the destination machine to which data is being sent. Accordingly, our model assumes wide-area networks with high data rates (10 to 40 Gbits) but sporadic packet loss, potentially bursty. The packet loss model used in our experiments is based on actual observations of TeraGrid, a scientific data network that links scientific supercomputing centers and has precisely these characteristics. In particular, Balakrishnan et al. [10] cite loss rates over 0.1% at times on uncongested optical-link paths between supercomputing centers. As a result, we emulate disaster with up to 1% loss rates in our evaluation of Section 5.

Of course, reliable transmission protocols such as TCP are typically used to communicate updates and acknowledgments between sites. Nonetheless, under our assumptions, a lost packet may prevent later received packets from being delivered to the mirrored storage system. The problem is that once the primary site has failed, there may be no way to recover a lost packet, and because TCP is sequenced, all data sent after the lost packet will be discarded in such situations — the gap prevents their delivery.

Data Loss Model: We consider data to be *lost* if an update has been acknowledged to the client, but the corresponding data no longer exists in the system. Today's remote mirroring regimes all experience data loss, but

the degree of disaster needed to trigger loss varies:

- Synchronous mirroring only sends acknowledgments to the client after receiving a response from the mirror. Data cannot be lost unless *both* primary and mirror sites fail.

- Semi-synchronous mirroring sends acknowledgments to the client after data written is locally stored at the primary site and an update is sent to the mirror. This scheme does not lose data unless the primary site fails *and* sent packets do not make it to the mirror. For example, packets may be lost while resident in local buffers and before being sent on the wire, the network may experience packet loss, partition, or components may fail at the mirror.

- Asynchronous mirroring sends acknowledgments to the client immediately after data is written locally. Data loss can occur even if just the primary site fails. Many products form snapshots periodically, for example, every twenty minutes [19, 31]. Twenty minutes of data could thus be lost if a failure disrupts snapshot transmission.

Goals: Our work can be understood as an enhancement of the semi-synchronous style of mirroring. The basic idea is to ensure that once a packet has been sent, the likelihood that it will be lost is as low as possible. We do this by sending error recovery data along with the packet and informing the sending application when error recovery has been sent. Further, by exposing link state, an error correcting coding scheme can be tuned to better match the characteristics observed in existing high-speed wide-area networks.

3 Network-Sync Remote Mirroring

Network-sync strikes a balance between performance and reliability, offering similar performance as semi-synchronous solutions, but with increased reliability. We use a forward-error correction protocol to increase the reliability of high-quality optical links. For example, a link that drops one out of every 1 trillion bits or 125 million 1 KB packets (this is the maximum error threshold beyond which current carrier-grade optical equipment shuts down) can be pushed into losing less than 1 out of every 10^{16} packets by the simple expedient of sending each packet twice — a figure that begins to approach disk reliability levels [7, 15]. By adding a callback when error recovery data has been sent, we can permit the application to resume execution once these encoded packets are sent, in effect treating the wide-area link as a kind of network disk. In this case, data is temporarily "stored" in the network while being shipped across the wide-area to the remote mirror. Figure 1 illustrates this capability.

One can imagine many ways of implementing this behavior (e.g. datacenter gateway routers). In general, implementations of network-sync remote mirroring must satisfy two requirements. First, they should *proactively* enhance the reliability of the network, sending recovery data without waiting for any form of negative acknowledgment (e.g. TCP fast retransmit) or timeouts keyed to the round-trip-time (RTT) to the remote site. Second, they must *expose* the status of outgoing data, so that the sender can resume activity as soon as a desired level of in-flight redundancy has been achieved for pending updates. Section 3.1 discusses the network-sync option, Section 3.2 discusses an implementation of it, and Section 3.3 discusses its tolerance to disaster.

3.1 Network-Sync Option

Assuming that an external client interacts with a primary site and the primary site implements some higher level remote mirroring protocol, network-sync enhances that remote mirroring protocol as follows. First, a host located at the primary site submits a write request to a local storage system such as a disk array (e.g. [12]), storage area network (e.g. [19]), or file server (e.g. [31]). The local storage system simultaneously applies the requested operation to its local storage image and uses a reliable transport protocol such as TCP to forward the request to a storage system located at the remote mirror. To implement the network-sync option, an egress router located at the primary site forwards the IP packets associated with the request, sends additional error correcting packets to an ingress router located at the remote site, and then performs a callback, notifying the local storage system which of the pending updates are now safely in transit[1]. The local storage system then replies to the requesting host, which can advance to any subsequent dependent operations. We assume that ingress and egress routers are under the control of site operators, thus can be modified to implement network-sync functionality.

Later, perhaps 50ms or so may elapse before the remote mirror storage system receives the mirrored request—possibly after the network-sync layer has reconstructed one or more lost packets using the combination of data received and error-recovery packets received. It applies the request to its local storage image, generates a storage level acknowledgment, and sends a response. Finally, when the primary storage system receives the response, perhaps 100ms later, it knows with certainty that the request has been mirrored and can garbage collect any remaining state (e.g. [19]). Notice that if a client requires the stronger assurances of a true remote-sync, the possibility exists of offering that guarantee selectively, on a per-operation basis. Figure 3 illustrates the network-sync mirroring option and Table 1 contrasts it to existing solutions.

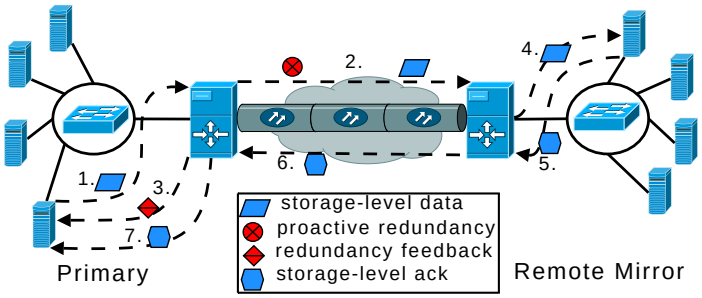

Figure 3: Network-Sync Remote Mirroring Option. (1) A primary-site storage system simultaneously applies a request locally and forwards it to the remote mirror. After the network-sync layer (2) routes the request and sends additional error correcting packets, it (3) sends an acknowledgment to the local storage system — at this point, the storage system and application can safely move to the next operation. Later, (4) a remote mirror storage system receives the mirrored request—possibly after the network-sync layer recovered some lost packets. It applies the request to its local storage image, generates a storage level acknowledgment, and (5) sends a response. Finally, (7) when the primary storage system receives the response, it knows with certainty that the request has been mirrored and can gargage collect.

Mirror Solution	Mirror Update	Mirror-ack Receive	Mirror-ack Latency	Rolling Disaster			
				Local-only Failure	Local + Pckt Loss	Local + NW Partition	Local+Mirror Failure
Local-sync	Async- or Semi-sync	N/A	N/A	Loss	Loss	Loss	Loss
Remote-sync	Synchronous	Storage-level ack (7)	WAN RTT	No loss	No loss	No loss	Maybe loss
Network-sync	Semi-sync	nw-sync feedback (3)	$\approx$ Local ping	$\approx$ No loss	$\approx$ No loss	Loss	Loss

Table 1: Comparison of Mirroring Protocols.

3.2 Maelstrom: Network-sync Implementation

The network-sync implementation used in our work is based on Forward Error Correction (FEC). FEC is a generic term for a broad collection of techniques aimed at proactively recovering from packet loss or corruption. FEC implementations for data generated in real-time are typically parameterized by a rate (r,c): for every r data packets, c error correction packets are introduced into the stream. Of importance here is the fact that FEC performance is independent of link length (except to the extent that loss rates may be length-dependent).

The specific FEC protocol we worked with is called Maelstrom [10], and is designed to match the observed loss properties of multi-hop wide-area networks such as TeraGrid. Maelstrom is a symmetric network appliance that resides between the datacenter and the wide-area link, much like a NAT box. The solution is completely transparent to applications using it, and employs a mixture of technologies: routing tricks to conceal itself from the endpoints, a link-layer reliability protocol (currently TCP), and a novel FEC encoding called *layered interleaving*, designed for data transfer over long-haul links with potentially bursty loss patterns. To minimize the rate-sensitivity of traditional FEC solutions, Maelstrom aggregates all data flowing between the primary and backup sites an operates on the resulting high-speed stream. See Balakrishnan et al. [10] for a detailed description of layered interleaving and analysis of its performance tolerance to random and bursty loss.

Maelstrom also adds feedback notification *callbacks*. Every time Maelstrom transmits a FEC packet, it also issues a callback. The local storage system then employs a redundancy model to infer the level of safety associated with in-flight data packets. For this purpose, a local storage system needs to know the underlying network's properties — loss rate, burst length, etc. It uses these to model the behavior of Maelstrom mathematically [10], and then makes worst-case assumptions about network loss to arrive at the needed estimate of the risk of data loss. We expect system operators monitor network behavior and periodically adjust Maelstrom parameters to adapt to any changes in the network characteristics.

There are cases in which the Maelstrom FEC protocol is unable to repair the loss (this can only occur if several packets are lost, and in specific patterns that prevent us from using FEC packets for recovery). To address such loss patterns, we run our mirroring solution over TCP, which in turn runs over Maelstrom: if Maelstrom fails to recover a lost packet, the end-to-end TCP protocol will recover it from the sender.

3.3 Discussion

The key metric for any disaster-tolerant remote mirroring technology is the distance by which datacenters can be separated. Today, a disturbing number of New York City banks maintain backups in New Jersey or Brooklyn, because they simply cannot tolerate higher latencies.

The underlying problem is that these systems typically operate over TCP/IP. Obviously, the operators tune the system to match the properties of the network. For example, TCP can be configured to use massive sender buffers and unusually large segments; also, an application can be modified to employ multiple side-by-side streams (e.g. GridFTP). Yet even with such steps, the protocol remains purely *reactive*—recovery packets are sent only in response to actual indications of failure, in the form of negative acknowledgments (i.e. fast retransmit) or timeouts keyed to the round-trip-time (RTT). Consequently, their recovery time is tightly linked to the distance between communicating end-points. TCP/IP, for example, requires a minimum of around 1.5 RTTs to recover lost data, which translates into substantial fractions of a second if the mirrors are on different continents. No matter how large we make the TCP buffers, the remote data stream will experience an RTT hiccup each time loss occurs: to deliver data in order, the receiver must await the missing data before subsequent packets can be delivered.

Network-sync evades this RTT issue, but does not protect the application against every possible rolling disaster scenario. Packets can still be queued in the local-area when disaster strikes. Further, the network can partitioned in the split second(s) before a primary site fails. Neither proactive redundancy or network-level callbacks will prevent loss in these cases. Accordingly, we envision that applications will need a mixture of remote-sync and network-sync, with the former reserved for particularly sensitive scenarios, and the latter used in most cases.

Another issue is failover and recovery. Since the network-sync option enhances remote mirroring protocols, we assume that a complete remote mirroring protocol will itself handle failover and recovery directly [19, 22, 20]. As a result, in this work, we focus on evaluating the fault tolerant capabilities of a network-sync option and do not discuss failover and recovery protocols.

4 Mirroring Consistency via SMFS

We will say that a mirror image is *inconsistent* if out of order updates are applied to the mirror, or the application updates a group of files, and a period ensues during which some of the mirrored copies reflect the updates but others are stale. Inconsistency is a well-known problem when using networks to access file systems, and the issue can be exacerbated when mirroring. For example, suppose that one were to mirror an NFS server, using the standard but unreliable UDP transport protocol. Primary and remote file systems can easily become inconsistent, since UDP packets can be reordered on the wire, particularly if a packet is dropped and the NFS protocol is forced to resend it. Even if a reliable transport protocol is used, in cases where the file system is spread over multiple storage servers, or applications update groups of files, skew in update behavior between the different mirrored servers may be perceived as an inconsistency by applications.

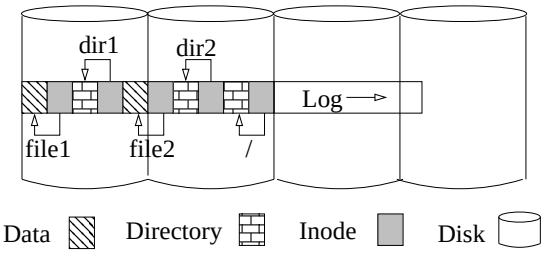

Figure 4: Format of a log after writing a file system with two sub directories `/dir1/file1` and `/dir2/file2`.

To address this issue, SMFS implements a file system that preserves the order of operations in the structure of the file system itself, a distributed log-structured file system (distributed-LFS)[2], where a particular log is distributed over multiple disks. Similar to LFS [35, 27], it embeds a UNIX tree-structured file system into an append only log format (Figure 4). It breaks a particular log into multiple segments that each have a finite maximum size and are the units of storage allocation and cleaning.

Although log-structured file systems may be unpopular in general settings (due to worries about high cleaning costs if the file system fills up), a log structure turns out to be nearly ideal for file mirroring. First, it is well known that an append-only log-structure is optimized for write performance [27, 35]. Second, by combining data and order of operations into one structure — the log — identical structures can be managed naturally at remote locations. Finally, log operations can be pipelined, increasing system throughput. Of course, none of this eliminates worries about segment cleaning costs. Our assumption is that because SMFS would be used only for files that need to be mirrored, such as backups and checkpoints, it can be configured with ample capacity—far from the tipping point at which these overheads become problematic.

In Sections 4.1 and 4.2, we describe the storage systems architecture and API.

4.1 SMFS Architecture

The SMFS architecture is illustrated in Figure 5. It works as follows. Clients access file system data by communi-

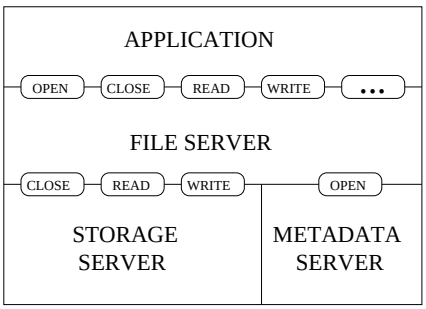

Figure 5: File System Architecture: Applications communicate with the *file server* through (possibly) a NFS interface. The file server in turn communicates with the *metadata server* through the `create()` function call. The metadata server allocates space for the newly created log on storage servers that it selects. The file server then interacts directly with the *storage server* for `append()`, `read()`, and `free()` operations.

cating with a file server (e.g. using the NFS protocol). File servers handle writes in a similar fashion to LFS. The log is updated by traversing a file system in a depth-first manner, first appending modified data blocks to the log, storing the log address in an inode, then appending the modified inode to the log, and so on [27, 29]. Reads are handled as in any conventional file system; starting with the root inode (stored in memory or a known location on disk) pointers are traversed to the desired file inode and data blocks. Although file servers provide a file system abstraction to clients, they are merely hosts in the storage system and stable storage resides with separate storage servers.

4.2 SMFS API

File servers interact with storage servers through a thin log interface—`create()`, `append()`, `read()`, and `free()`. `create()` communicates with a metadata server to allocate storage resources for a new log; it assigns responsibility for the new log to a storage server. After a log has been created, a file server uses the `append()` operation to add data to the log. The file server communicates directly with a log's storage server to append data. The storage server assigns the order of each append—assigns the address in the log to a particular append—and atomically commits the operation. SMFS maintains group-mirroring consistency, in which a single `append()` can contain updates to many different files where the group of updates will all be reflected by the storage system atomically, either all or none. `read()` returns the data associated with a log address. Finally, `free()` takes a log address and marks the address for later cleaning. In particular, after a block has been modified or file removed, the file system calls `free()` on all blocks that are no longer referenced. The `create()`, `append()`, and `free()` operations are mirrored between the primary site and remote mirror.

5 Evaluation

In this section, we evaluate the network-sync remote mirroring option, running our SMFS prototype on Emulab [40] and the Cornell National Lambda Rail (NLR) Rings testbed [1].

5.1 Experimental Environment

The implementation of SMFS that we worked was implemented as a user-mode application coded in Java. SMFS borrows heavily from our earlier file system, Antiquity [39]; however, the log address was modified to be a segment identifier and offset into the segment. A hash of the block can optionally be computed, but it is used as a checksum instead of as part of the block address in the log. We focus our evaluation on the `append()` operation since that is by far the dominant operation mirrored between two sites.

SMFS uses the Maelstrom network appliance [10] as the implementation of the network-sync option. Maelstrom can run as a user-mode module, but for the experiments reported here, it was dropped into the operating system, where it runs as a Linux 2.6.20 kernel module with hooks into the kernel packet filter [2]. Packets destined for the opposite site are routed through a pair of Maelstrom appliances located at each site. More importantly, situating a network appliance at the egress and ingress router for each site creates a virtual link between the two sites, which presents many opportunities for increasing mirroring reliability and performance.

The Maelstrom egress router captures packets, which it processes to create redundant packets. The original IP packets are forwarded unaltered; the redundant packets are then sent to the ingress router using a UDP channel. The ingress router captures and stores a window consisting of the last K IP packets that it has seen. Upon receiving a redundant packet it checks it against the last K IP packets. If there is an opportunity to recover any lost IP packet it does so, and forwards the newly recovered IP packet through a raw socket to the intended destination. Note that each appliance works in both egress and ingress mode since we handle duplex traffic.

To implement network-sync redundancy feedback, the Maelstrom kernel module tracks each TCP flow and sends an acknowledgment to the sender. Each acknowledgment includes a byte offset from the beginning of the stream up to the most recent byte that was included in an error correcting packet that was sent to the ingress router.

We used the TCP Reno congenstion control algorithm to communicate between mirrored storage systems for all experiments. We experimented with other congestion control algorithms such as cubic; however, the results

were nearly identical since we were measuring packets lost after a primary site failure due to a disaster.

We tested the setup on Emulab [40]; our topology emulates two clusters of eight machines each, separated by a wide-area high capacity link with 50 to 200 ms RTT and 1 Gbps. Each machine has one 3.0 GHz Pentium 64-bit Xeon processor with 2.0 GB of memory and a 146 GB disks. Nodes are connected locally via a gigabit Ethernet switch. We apply load to these deployments using up to 64 testers located on the same cluster as the primary. A single tester is an individual application that has only one outstanding request at a time. Figure 3 shows the topology of our Emulab experimental setup (with the difference that we used eight nodes per cluster, and not four). Throughout all subsequent experiments, link loss is random, independent and identically distributed. See Balakrishnan et al [10] for an analysis with bursty link loss. Finally, all experiments show the average and standard deviation over five runs.

The overall SMFS prototype is fast enough to saturate a gigabit wide-area link, hence our decision to work with a user-mode Java implementation has little bearing on the experiments we now report: even if SMFS was implemented in the kernel in C, the network would still be the bottleneck.

5.2 Evaluation Metrics

We identify the following three metrics to evaluate the efficacy of SMFS:

- **Data Loss:** What happens in the event of a disaster at the primary? For varying loss rates on the wide-area link, how much does the mirror site diverge from the primary? We want our system to minimize this divergence.

- **Latency:** Latency can be used to measure both performance and reliability. Application-perceived latency measures (perceived) performance. Mirroring latency, on the other hand, measures reliability. In particular, the lower the latency, and the smaller the spread of its distribution, the better the fidelity of the mirror to the primary.

- **Throughput:** Throughput is a good measure of performance. The property we desire from our system is that throughput should degrade gracefully with increasing link loss and latency. Also, for mirroring solutions that use forward error correcting (FEC) codes, there is a fundamental tradeoff between data reliability and goodput (i.e. application level throughput); proactive redundancy via FEC increases tolerance to link loss and latency, but reduces the maximum goodput due to the overhead of FEC codes. We focus on goodput.

Layered Interleaving FEC Params[10]	r	8
	c	3
Network-sync Parameters	segment size	100 MB
	append size	512 kB
	block size	4 kB
Experiment Parameters	expt duration	3 mins

Table 2: Experimental Configuration Parameters

For effective comparison, we define the following five configurations; all configurations use TCP to communicate between each pair of storage servers.

- **Local-sync:** This is the canonical state-of-the-art solution. It is a semi-synchronous solution. As soon as the request has been applied to the local storage image and the local kernel buffers a request to send a message to the remote mirror, the local storage server responds to the application; it does not wait for feedback from remote mirror, or even for the packet to be placed on the wire.

- **Remote-sync:** This is the other end of the spectrum. It is a synchronous solution. The local storage server waits for a storage-level acknowledgment from the remote mirror before responding to the application.

- **Network-sync:** This is SMFS running with a network-sync option, implemented by Maelstrom in the manner outlined in Section 3 (e.g. with TCP over FEC). The network-sync layer provides feedback after proactively injecting redundancy into the network. SMFS responds to the application after receiving these redundancy notification.

- **Local-sync+FEC:** As a comparison point, this scheme is the local-sync mechanism, with Maelstrom running on the wide-area link, but without network-level callbacks to report when FEC packets are placed on the wire (i.e. storage servers are unaware of the proactive redundancy). The local server permits the application to resume execution as soon as data has been written to the local storage system.

- **Remote-sync+FEC:** As a second comparison point, this scheme is the remote-sync mechanism, again using Maelstrom on the wide-area link but without upcalls when FEC packets are sent. The local server waits for the remote storage system to acknowledge updates.

These five SMFS configurations are evaluated on each of the above metrics, and their comparative performance is presented. The Network-sync, Local-sync+FEC, and

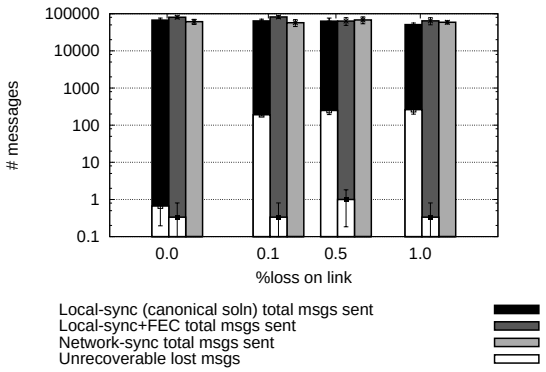

Figure 6: Data loss as a result of disaster and wide-area link failure, varying link loss (50ms one-way latency and FEC params $(r,c) = (8,3)$).

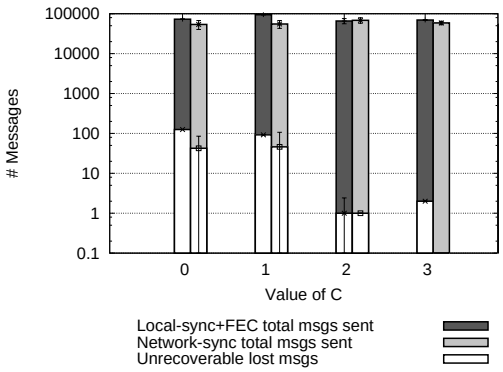

Figure 7: Data loss as a result of disaster and wide-area link failure, varying FEC param c (50ms one-way latency, 1% link loss).

Remote-sync+FEC configurations all use the Maelstrom layered interleaving forward error correction codes with parameters $(r,c) = (8,3)$, which increases the tolerance to network transmission errors, but reduces the goodput by as much as $8/11$ of the maximum throughput without any proactive redundancy. Table 2 lists the configuration parameters used in the experiments described below.

5.3 Reliability During Disaster

We measure reliability in two ways:

- In the event of a disaster at the primary site, how much data loss results?
- How much are the primary and mirror sites allowed to diverge?

These questions are highly related; we distinguish between them as follows: The maximum amount by which the primary and mirror sites can diverge is the extent of the bandwidth-delay product of the link between them; however, the amount of data lost in the event of failure depends on how much of this data has been acknowledged to the application. In other words, how often can we be caught in a lie? For instance, with a remote-sync solution (synchronous mirroring), though bandwidth-delay product – and hence primary-to-mirror divergence – may be high, data loss is zero. This, of course, is at severe cost to performance. With a local-sync solution (async- or semi-synchronous mirroring), on the other hand, data loss is equal to divergence. The following experiments show that the network-sync solution with SMFS achieves a desirable mean between these two extremes.

Disaster Test Figure 6 shows the amount of data loss in the event of a disaster for the local-sync, local-sync+FEC, and network-sync solutions; we do not test the remote-sync and remote-sync+FEC solutions in this experiment since these solutions do not lose data.

The rolling disaster, failure of the wide-area link and crash of all primary site processes, occurred two minutes into the experiment. The wide-area link operated at 0% loss until immediately before the disaster occurred, when loss rate was increased for 0.5 seconds, thereafter the link was killed (See Section 2 for a description of rolling disasters). The x-axis shows the wide-area link loss rate immediately before the link is killed; link losses are random, independent and identically distributed. The y-axis shows both the total number of messages sent and total number of messages lost—lost messages were perceived as durable by the application but were not received by the remote mirror. Messages were of size 4 kB.

The total number of messages sent is similar for all configurations since the link loss rate was 0% for most of the experiment. However, local-sync lost a significant number of messages that had been reported to the application as durable under the policy discussed in Section 3.1. These unrecoverable messages were ones buffered in the kernel, but still in transit on the wide area link; when the sending datacenter crashed and the link (independently) dropped the original copy of the message, TCP recovery was unable to overcome the loss.

Local-sync+FEC lost packets as well: it lost packets still buffered in the kernel, but not packets that had already been transmitted — in the latter case, the proactive redundancy mechanism was adequate to overcome the loss. The best outcome is visible in the right-most histogram at 0.1%, 0.5%, and 1% link loss: here we see that although the network-sync solution experienced the same level of link-induced message loss, all the lost packets that had been reported as durable to the sender application were in fact recovered on the receiver side of the link. This supports the premise that a network-sync solution can tolerate disaster while minimizing loss.

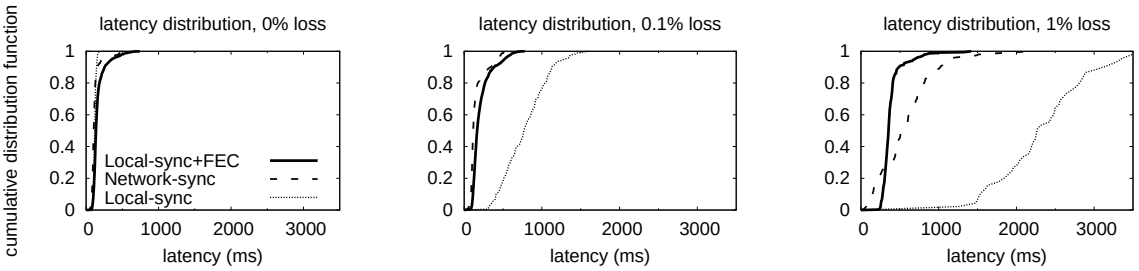

Figure 8: Latency distribution as a function of wide-area link loss (50ms one-way latency).

Combined with results from Section 5.4, we demonstrate that the network-sync solution actually achieves the best balance between reliability and performance.

Figure 7 quantifies the advantage of network-sync over local-sync+FEC. In this experiment, we run the same disaster scenario as above, but with 1% link loss during disaster and we vary the FEC parameter c (i.e. the number of recovery packets). At $c = 0$, there are no recovery packets for either local-sync+FEC or network-sync—if a data packet is lost during disaster, it cannot be recovered and TCP cannot deliver any subsequent data to the remote mirror process. Similarly, at $c = 1$, the number of lost packets is relatively high for both local-sync+FEC and network-sync since one recovery packet is not sufficient to mask 1% link loss. With $c > 1$, the number of recovery packets is often sufficient to mask loss on the wide-area link; however, local-sync+FEC loses data packets that did not transit outside the local-area before disaster, whereas with network-sync, primary storage servers respond to the client only after receiving a callback from the egress gateway. As a result, network-sync can potentially reduce data loss in a disaster.

Latency Figure 8 shows how latency is distributed across all requests for local-sync, local-sync+FEC, and network-sync solutions. Latency is the time between a local storage server sending a request and a remote storage server receiving the request. We see that these solutions show similar latency for zero link loss, but local-sync+FEC and network-sync show considerably better latency than local-sync for a lossy link. Furthermore, the latency spread of local-sync+FEC and network-sync solutions is considerably less than the spread of the local-sync solution — particularly as loss increases; proactive redundancy helps to reduce latency jitter on lossy links. Smaller variance in this latency distribution helps to ensure that updates submitted as a group will arrive at the remote site with minimum temporal skew, enabling the entire group to be written instead of not.

5.4 Performance

System Throughput Figure 9 compares the performance of the five different mirroring solutions. The x-axis shows loss probability on the wide-area link being

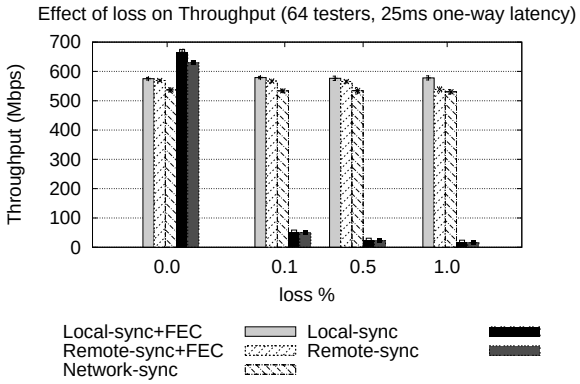

Figure 9: Effect of varying wide-area one-way link *loss* on Aggregate Throughput.

increased from 0% to 1%, while the y-axis shows the throughput achieved by each of these mirroring solutions. All mirroring solutions use 64 testers over eight storage servers.

At 0% loss we see that the local-sync and remote-sync solutions achieve the highest throughput because they do not use proactive redundancy, thus the goodput of the wide-area link is not reduced by the overhead of any forward error correcting packets. On the other hand, local-sync+FEC, remote-sync+FEC, and network-sync achieve lower throughput because the forward error correcting packets reduce the goodput in these cases. The forward error correction overhead is tunable; increasing FEC overhead often increases transmission reliability but reduces throughput. There is a slight degradation of performance for network-sync since SMFS waits for feedback from the egress router instead of responding immediately after the local kernel buffers the send request. Finally, the remote-sync and remote-sync+FEC achieve comparable performance to all the other configurations since there is no loss on the wide-area link and the storage servers can saturate the link with overlapping mirroring requests.

At higher loss rates, 0.1%, 0.5%, and 1%, we see that any solution that uses proactive redundancy (local-sync+FEC, remote-sync+FEC, and network-sync) achieves more than an order of magnitude higher

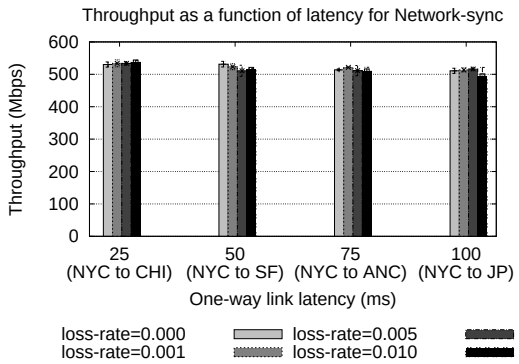

Figure 10: Effect of varying wide-area link *latency* on Aggregate Throughput.

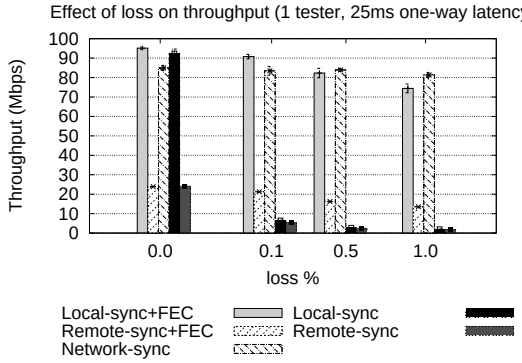

Figure 11: Effect of varying wide-area link *loss* on Per-Client Throughput.

throughput over any solution that does not. This illustrates the power of proactive redundancy, which makes it possible for these solutions to recover from lost packets at the remote mirror using locally-available data. Further, we observe that these proactive redundancy solutions perform comparably in both asynchronous and synchronous modes: in these experiments, the wide-area network is the bottleneck since overlapping operations can saturate the wide-area link.

Figure 10 shows the system throughput of the network-sync solution as the wide-area one-way link latency increases from 25 ms to 100 ms. It demonstrates that the network-sync solution (or any solution that uses proactive redundancy) can effectively mask latency and loss of a wide-area link.

Application Throughput The previous set of experiments studied system-level throughput, using a large number of testers. An interesting related study is presented here, of individual-application throughput in each SMFS configuration. Figure 11 shows the effect of increasing loss probability on the throughput of a application, with only one outstanding request at a time.

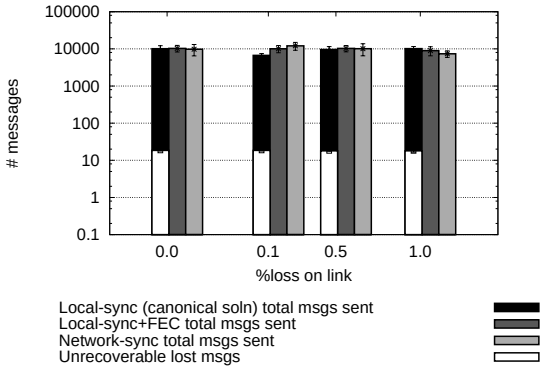

Figure 12: Data loss as a result of disaster and wide-area link failure (Cornell NLR-Rings, 37 ms one-way delay).

We see now that local-sync(+FEC) and network-sync solutions perform better than remote-sync(+FEC). The reason for this difference is that with asynchrony, network-sync can return an acknowledgment to the application as soon as a request is on the wide-area link, providing an opportunity to pipeline requests. This is in contrast to conventional asynchrony, where the application would receive an acknowledgment as soon as a request is *buffered*. The advantage with the former is that it provides performance gain without hurting reliability. The disadvantage is that pure buffering is a local system call operation, which can return to the application sooner and can achieve higher throughput as seen by the local-sync(+FEC) solutions. However, this increase in throughput is at a sacrifice of reliability; any buffered data may be lost in the event of a crash before it is sent (See Figure 6).

5.5 Cornell National Lambda Rail Rings

In addition to our emulated setup and results, we are beginning to physically study systems that operate on dedicated lambda networks that might be seen in cutting edge financial, military, and educational settings. To study these "personal" lambda networks, we have created a new testbed consisting of optical network paths of varying physical length that start and end at Cornell, the Cornell National Lambda Rail (NLR) Rings testbed.

The Cornell NLR-Rings testbed consists of three rings: a *short* ring that goes from Cornell to New York City and back, a *medium* ring that goes to Chicago down to Atlanta and back, and a *long* ring that goes to Seattle down to Los Angeles and back. The one-way latency is 7.9 ms, 37 ms, and 94 ms, for the short, medium, and long rings, respectively. The underlying optical networking technology is state-of-the-art: a 10 Gbps wide-area network running on dedicated fiber optics (separate from the public Internet) and created as a scientific research infrastructure by the NLR consortium [3]. Each ring

includes multiple segments of optical fiber, linked by routers and repeaters. More importantly, for the medium and long ring, each network packet traverses a unique path without going along the same segment. See NLR [3] for a map.

Though all rings in the testbed are capable of 10 Gbps end-to-end, we are only able to operate at hundreds of megabits per second at this time due to network construction. Nonetheless, we are able to study the effects of disaster on dedicated wide-area lambda networks and hope to be able to use increasingly more bandwidth in the future.

To study the effects of disaster in this wide-area testbed, we conduct the same disaster experiment described in Section 5.3. We induced loss on the wide-area link 0.5 second before the primary site fails via a router that we control. Later, when the primary site fails, the wide-area link and all processes are killed. Figure 12 shows data loss during this disaster for the medium path on the Cornell NLR-Rings testbed. The x-axis shows the loss induced on the wide-area link (link losses are random, independent and identically distributed) and the y-axis shows the number of messages sent and the number of unrecoverable messages. There are two interesting results illustrated. First, local-sync lost messages even when no loss was induced on the wide-area link. This may be because our wide-area testbed may drop packets, which prevents local-sync protocols from delivering to the mirroring application. Local-sync+FEC and network-sync, on the other hand, did not lose messages because both can mask wide-area link loss. Second, due to the relatively low bandwidth, packets were able to transit outside of the local-area, preventing loss from occurring in the local-area and enabling both local-sync+FEC and network-sync to mask wide-area link loss.

6 Related Work

6.1 Mirroring modes

Synchronous mirroring, like IBM's Peer-to-Peer Remote Copy (PPRC) [6] and EMC's Symmetrix Remote Data Facility (SRDF) [12] is a technique often used in disaster tolerance solutions. It guarantees that local copies of data are consistent with copies at a remote site, and also guarantees that the mirror sites are as up-to-date as possible. Naturally, the drawback is that of added I/O latency to every write operation; furthermore, long distance links make this technique prohibitively expensive.

An alternate solution is to use asynchronous remote mirroring [19, 24, 31]. For example, SnapMirror [31] provides asynchronous mirroring of file systems by periodically transferring self-consistent data snapshots from a source volume to a destination volume. Users are provided with a knob for setting the frequency of updates — if set to a high value, the mirror would be nearly current with the source, while setting to a low value reduces the network bandwidth consumption at the risk of increased data loss. Seneca [19] is a storage area network mirroring solution and similarly attempts to reduce the amount of traffic sent over the wide-area network.

SnapMirror works at the block level, using the WAFL [17] file system active block map to identify changed blocks and avoid sending deleted blocks. Moreover, since it operates at this level, it is able to optimize data reads and writes. The authors showed that for update intervals as short as one minute, data transfers were reduced by 30% to 80%.

Similar to SnapMirror, Seneca [19] is another asynchronous mirroring solution that attempts to reduce the traffic sent over the wide-area network, but also increases the risk of data loss. Seneca operates at the level of a storage area network (SAN) instead of the file system level.

Semi-synchronous mirroring is yet another mode of operation, closely related to both synchronous and asynchronous mirroring. In such a mode, writes are sent to both the local and the remote storage sites at the same time, the I/O operation returning when the local write is completed. However subsequent write I/O is delayed until the completion of the preceding remote write command. In [42] the authors show that by leveraging a log policy for the active remote write commands the system is able to allow a limited number of write I/O operations to proceed before waiting for acknowledgment from the remote site, thereby reducing the latency significantly.

6.2 Error correcting codes

Packet level forward error correcting (FEC) schemes typically transmit c repair packets for every r data packets, using coding schemes with which all data packets can be reconstructed if at least r out of $r+c$ data and repair packets are received [18]. In contrast, convolution codes work on bit or symbol streams of arbitrary length, and are most often decoded with the Viterbi algorithm [38]. Our work favors FEC: FEC schemes have the benefit of being highly tunable – trading off overhead and timeliness, and are very stable under stress – provided that the recovery does not result in high levels of traffic.

FEC techniques are increasingly popular. Recent applications include FEC for multicasting data to large groups [34], where FEC can be employed either by receivers [9] or senders [18, 28]. In general, fast, efficient encodings like Tornado codes [11] make sender-based FEC schemes very attractive in scenarios where dedicated senders distribute bulk data to a large number of receivers.

Likewise, FEC can be used when connections experience long transmission delays, in which case the use of

redundancy helps bound the delivery delays within some acceptable limits, even in the presence of errors [18, 33]. For example, deep space satellite communications [43] have been using error correcting codes for decades both for achieving maximal information transfer over a restricted bandwidth communication link and in the presence of data corruption.

SMFS is not the first system to propose exposing network state to higher level storage systems [32]. The difference, however, is that network-sync can be implemented with gateway routers under the control of site operators and does not require change to wide-area Internet routers.

6.3 Reliable Storage & Recovery

Recent studies have shown that failures plague storage and other components of large computing datacenters [36]. As a result, many systems replicate data to reduce risk of data loss [5, 14, 16, 25, 23, 37]. However, replication alone is not complete without recovery.

Recovery in the face of disaster has been a problem that has received a lot of attention [13, 21, 22]. In [20], for example, the authors propose a reactive way to solve the data recovery scheduling problem once the disaster has occurred. Potential recovery processes are first mapped onto recovery graphs — the recovery graphs capture alternative approaches for recovering workloads, precedence relationships, timing constraints, etc. The recovery scheduling problem is encoded as an optimization problem with the end goal of finding the schedule that minimizes some measure of penalty; several methods for finding optimal and near-optimal solutions are given.

Aguilera et. al. [4] explore the tradeoff between the ability to recover and the cost of recovery in enterprise storage systems. They propose a multi-tier file system called TierFS that employs a "recoverability log" used to increase the recoverability of lower tiers by using the highest tier.

Both LOCKSS [26] and Deep Store [44] address the problem of reliably preserving large volumes of data for virtually indefinite periods of time, dealing with threats like format obsolescence and "bit-rot." LOCKSS consists of a set of low-cost, independent, persistent cooperating caches that use a voting scheme to detect and repair damaged content. Deep Store eliminates redundancy both within and across files; it distributes data for scalability and provides variable levels of replication based on the importance or the degree of dependency of each chunk of stored data.

Baker et. al. [8] consider the problem of recovery from failure of long-term storage of digital information. They propose a "reliability model" encompassing latent and correlated faults, and the detection time of such latent faults. They show that a simple combination of auditing (to detect latent faults) as soon as possible, automatic recovery and independence of replicas yields the most benefit with respect to the cost of each technique.

7 Conclusion

The conundrum facing many disaster tolerance and recovery designs is the tradeoff between loss of performance and the potential loss of data. On the one hand, it may not be desirable to slow application response time until it is assured that data will not be lost in the event of disaster. On the other hand, the prospect of data loss can be catastrophic for many companies and organizations. Unfortunately, there is not much of a middle ground in the design space and designers must choose one or the other.

The network-sync remote mirroring option potentially offers an improvement, providing performance of enterprise-level semi-synchronous remote mirroring solutions while increasing their data reliability guarantees. Like native semi-synchronous protocols, network-sync protocols simultaneously send each update to the remote mirror as the primary handles the update locally. Rather than waiting for an acknowledgment from the remote mirror, it delays only until it receives feedback from an underlying communication layer, acknowledging that data and repair packets have been placed on the external wide-area network. This minimizes the loss of data in the event of disaster. Applications requiring strong remote-sync guarantees can still wait for a remote acknowledgment, but for most purposes, network-sync represents an appealing new option. Our experiments show that SMFS, a remote mirroring solution that uses the network-sync option, exhibits performance that is independent of link-latency, in marked contrast to most existing technologies.

Acknowledgments

We would like to thank our shepherd James Plank, the anonymous reviewers, and Robbert van Renesse for their comments that shaped the final version of this paper. Also, we would like to thank all who contributed to setting up the Cornell NLR-Rings testbed: Dan Freedman, Cornell Facilities Support Scott Yoest and Larry Parmelee, CIT-NCS networking engineering Eric Cronise and Dan Eckstrom, and NLR network engineering Greg Boles, Brent Sweeny, and Joe Lappa. Finally, we would like to thank Intel and Cisco for providing necessary routing and computing equipment, and NSF TRUST and AFRL Castor grant for funding support.

Notes

[1] Egress and ingress routers operate as gateway routers between datacenter and wide-area networks, where egress routers send packets from local datacenter networks to the wide-area network and ingress

routers receive packets from the wide-area network and forward packets to local datacenter networks. Generally, egress routers also function as ingress routers and visa versa since they handle duplex traffic.

[2] A distributed log-structured file system can expose an NFS interface to hosts; however, it stores data in a distributed log-structured file system instead of a local UNIX file system (UFS).

References

[1] Cornell national lambda rail (nlr) rings testbed. `http://www.cs.cornell.edu/~hweather/nlr`.

[2] Firewalling, nat, and packet mangling for linux. `http://www.netfilter.org`. Last accessed Jan. 2009.

[3] National lambda rail. `http://www.nlr.net`.

[4] AGUILERA, M. K., KEETON, K., A, M., MUNISWAMY-REDDY, K., AND UYSAL, M. Improving recoverability in multi-tier storage systems. In *Proc. of DSN* (2007).

[5] ANDERSON, T., DAHLIN, M., NEEFE, J., PATTERSON, D., ROSELLI, D., AND WANG, R. Serverless Network File Systems. In *Proc. of ACM SOSP* (Dec. 1995).

[6] AZAGURY, A., FACTOR, M., AND MICKA, W. Advanced functions for storage subsystems: Supporting continuous availability. An IBM SYSTEM Journal, 2003.

[7] BAIRAVASUNDARAM, L., GOODSON, G., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *Proc. of ACM SIGMETRICS* (June 2007).

[8] BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALE, P. A fresh look at the reliability of long-term digital storage. *SIGOPS Oper. Syst. Rev. 40*, 4 (2006), 221–234.

[9] BALAKRISHNAN, M., BIRMAN, K., PHANISHAYEE, A., AND PLEISCH, S. Ricochet: Lateral error correction for time-critical multicast. In *NSDI* (Apr. 2007).

[10] BALAKRISHNAN, M., MARIAN, T., BIRMAN, K., WEATHERSPOON, H., AND VOLLSET, E. Maelstrom: Transparent error correction for lambda networks. In *NSDI* (2008).

[11] BYERS, J. W., LUBY, M., MITZENMACHER, M., AND REGE, A. A digital fountain approach to reliable distribution of bulk data. *SIGCOMM Comput. Commun. Rev. 28*, 4 (1998), 56–67.

[12] CORP, E. Symmetrix remote data facility. `http://www.emc.com/products/family/srdf-family.htm`. Last accessed Jan. 2009.

[13] COUGIAS, D., HEIBERGER, E., AND KOOP, K. The backup book: disaster recovery from desktop to data center. Schaser-Vartan Books, Lecanto, FL, 2003.

[14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The google file system. In *Proc. of ACM SOSP* (Oct. 2003), pp. 29–43.

[15] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving file system reliability with i/o shepherding. In *Proc. of ACM SOSP* (Oct. 2007).

[16] HARTMAN, J. H., AND OUSTERHOUT, J. K. The zebra striped network file system. *ACM TOCS* (1995).

[17] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference* (1994).

[18] HUITEMA, H. The case for packet level FEC. In *Proc. of the IFIP Workshop on Protocols for High-Speed Networks* (Oct. 1996).

[19] JI, M., VEITCH, A., AND WILKES, J. Seneca: Remote mirroring done write. In *Proc. of USENIX FAST* (June 2003).

[20] KEETON, K., BEYER, D., BRAU, E., MERCHANT, A., SANTOS, C., AND ZHANG, A. On the road to recovery: Restoring data after disasters. In *Proc. of ACM EuroSys* (Apr. 2006).

[21] KEETON, K., AND MERCHANT, A. A framework for evaluating storage system dependability. In *Proc. of DSN* (Washington, DC, USA, 2004), IEEE Computer Society, p. 877.

[22] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for disasters. In *Proc. of USENIX FAST* (Berkeley, CA, USA, 2004), USENIX Association, pp. 59–62.

[23] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *Proc. of ACM ASPLOS* (1996), pp. 84–92.

[24] LEUNG, S. A., MACCORMICK, J., PERL, S. E., AND ZHANG, L. Myriad: Cost-effective disaster tolerance. In *Proc. of USENIX FAST* (Jan. 2002).

[25] LISKOV, B., GHEMAWAT, S., GRUBER, R., JOHNSON, P., SHRIRA, L., AND WILLIAMS, M. Replication in the harp file system. In *Proc. of ACM SIGOPS* (1991).

[26] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM TOCS* (2005).

[27] MATTHEWS, J., ROSELLI, D., COSTELLO, A., WANG, R., AND ANDERSON, T. Improving the performance of log-structured file systems with adaptive methods. In *Proc. of ACM SOSP* (1997).

[28] NONNENMACHER, J., BIERSACK, E. W., AND TOWSLEY, D. Parity-based loss recovery for reliable multicast transmission. *IEEE/ACM Transactions on Networking 6*, 4 (1998), 349–361.

[29] ORACLE. Berkeley db java edition architecture. An Oracle White Paper, Sept. 2006. http://www.oracle.com/database/ berkeley-db/je/index.html.

[30] PARSONS, D. S., PERETTI, B., AND GROCHOW, J. M. Closing the gap: A research and development agenda to improve the resiliency of the banking and finance sector. In *U.S. Department of the Treasury Study* (Mar. 2005).

[31] PATTERSON, H., MANLEY, S., FEDERWISCH, M., HITZ, D., KLEIMAN, S., AND OWARA, S. Snapmirror: File system based asynchronous mirroring for disaster recovery. In *Proc. of USENIX FAST* (Jan. 2002).

[32] PLANK, J. S., BASSI, A., BECK, M., MOORE, T., SWANY, D. M., AND WOLSKI, R. Managing data storage in the network. *IEEE Internet Computing 5*, 5 (September/October 2001), 50–58.

[33] RIZZO, L. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review* (1997).

[34] RIZZO, L., AND VICISANO, L. RMDP: An fec-based reliable multicast protocol for wireless environments. In *ACM SIGCOMM Mobile Computer and Communication Review* (New York, NY, USA, 1998), vol. 2, ACM Press, pp. 22–31.

[35] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[36] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proc. of USENIX FAST* (2007).

[37] THEKKATH, C., MANN, T., AND LEE, E. Frangipani: A scalable distributed file system. In *Proc. of ACM SOSP* (1997).

[38] VITERBI, A. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IT 13* (1967), 260–269.

[39] WEATHERSPOON, H., EATON, P., CHUN, B., AND KUBIATOWICZ, J. Antiquity: Exploiting a secure log for wide-area distributed storage. In *Proc. of ACM EuroSys* (2007).

[40] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proc. of USENIX OSDI* (2002).

[41] WITTY, R. J., AND SCOTT, D. Disaster recovery plans and systems are essential. *Gartner Research Gartner FirstTake: FT-14-5021* (September 12 2001).

[42] YAN, R., SHU, J., AND CHAN WEN, D. An implementation of semi-synchronous remote mirroring system for sans. In *ACM Workshop of Grid and Cooperative Computing (GCC)* (2004).

[43] YEUNG, R. W., AND ZHANG, Z. Distributed source coding for satellite communications. *IEEE Transactions on Information Theory 45*, 4 (May 1999), 1111–1120.

[44] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep store: an archival storage system architecture. In *Proc. 21st International Conference on Data Engineering (ICDE)* (Apr. 2005).