# *e*-SAFE: An Extensible, Secure and Fault Tolerant Storage System

Arnab Paul Sandip Agarwala Umakishore Ramachandran
College of Computing

Georgia Tech
Atlanta, GA 30332, USA
{arnab, sandip, rama}@cc.gatech.edu

## Abstract

*With the rapidly falling price of hardware, and increasingly available bandwidth, the storage technology is seeing a paradigm shift from centralized and managed mode to distributed and un-managed configurations. The key issues in designing such system include scalability, extensibility and robustness to name a few.*

*This paper describes e-SAFE , a scalable distributed storage system that deploys a pastiche of theoretical and practical techniques, providing tolerance of malicious faults, reduced management overhead such as periodic repairs, and very high availability at an archival scale. e-SAFE is designed to provide a storage utility for environments such as large-scale data centers in enterprise networks where the servers experience high loads and thus show temporary unavailability (as opposed to P2P systems, where servers disappear over the long run). Consequently, the design goals of e-SAFE is to provide high load resilience in a seamlessly extensible way. e-SAFE is based on the simple principle: efficiently sprinkle data all over a distributed storage and robustly reconstruct even when many of them are unavailable under high loads.*

*The performance gears used in e-SAFE are:* (i) *Task parallelization over multiple file segments that can take advantage of an SMP architecture,* (ii) *Erasure codes with very fast encoding and decoding algorithms as opposed to naive replications and* (iii) *A back-ground replication mechanism hiding the cost of replication and dissemination from the user, yet guaranteeing high durability.*

## 1 Introduction

In the context of a large scale distributed storage, we ask the question - *How to design a storage architecture that is robust, scalable, extensible, highly available and that delivers high performance*? The main motivation behind our query is storage technology's perceivable shift from centrally managed data servers to distributed units. Consider an organization such as a corporate house or a university that has a large set of machines; the collected capacity of their local disk-spaces far exceeds those of the typical file servers dedicated to host data for them. Moreover, it has become evident, that for storage, cost of management will strictly dominate that of the hardware. Thus the new paradigm is geared toward reducing cost of ownership of data [12]. While the immediate benefit is robustness, an economic impact is reduced management overhead; since information would be efficiently replicated and scattered into many packets, one need not worry about a few (or possibly many) packets that may be lost due to hardware or software malfunctions.

In this paper, we describe *e*-SAFE , a robust, storage system that is tailored to the requirements of this upcoming paradigm. In the process of developing *e*-SAFE , we borrowed from a spectrum of design principles, both theoretical and engineering, and deployed them coherently into a single system architecture. We focus on a very large scale storage system that is quite common in large organizations. Such hardware infrastructures typically grow very fast in size, are bound to work under very high loads and are required to provide high availability to the data that is stored within. Consequently our design goals differ substantially from a P2P like file sharing system. However, our design principles borrow from the insights gained through research in the space of distributed and P2P computing.

#### Enabling Technologies

For quite a few years now, researchers have been investigating the design of massively large scale storage systems that can potentially span the internet. Paradigms such as Peer-to-Peer systems and the Grid computing are bringing diverse computing domains within cooperative environments in order to harness and utilize computing resources more effectively. While traditional P2P systems have been explored to carve out storage-utilities [39, 22, 36], a significant research has been done in enabling storage-area-networks(SAN) that can truly span large geographical distributions [44, 45]. Thus traditional storage systems have been evolving to become internet-wide with IP based underlying communication subsystems (IPSAN [32]).

### Key Issues

Irrespective of the organizational specifics, the design of any large distributed storage encompasses a few key issues: *Scalability, Availability, Integrity and Security*. As systems scale up, component failures become more of a norm than an isolated event. Thus scaling up demands robustness. Our specific goals for the design of a robust storage system include reducing management overhead that manifests itself in two forms: (*i*) *Repair overhead* - In the face of failures a monitoring unit has to keep track of any data loss and follow up with appropriate recovery as well as create regular back-ups, and (*ii*) *Handling Extensibility* - because of the rapidly falling price of disks, it is easy to conceive that storage system of an organization will see rapid growth with time, enabling further reduction in cost of ownership for individual storage units. So, from a management point of view, while it is lucrative to further reduce the cost of ownership by creating additional redundancy over the *extension* units, it is also essential that such readjustment be seamless and fast. Traditionally, replication or erasure codes are used for these purposes. While the former is extremely space inefficient, the latter, *i.e.,* the traditional erasure codes are limited in a number of ways to incorporate scaling up; explicit parameter tunings are necessary. Added to that are algorithmic performance penalties.

### *e*-SAFE

*e*-SAFE provides seamless extensibility, tolerance of malicious faults, reduced management overhead such as periodic repairs, and very high availability at an archival scale. The design of *e*-SAFE rests upon a few key principles: (*i*) Use of a specific class of erasure codes called the *Fountain Codes* for seamless extensibility and fast encoding/decoding, and (*ii*) Efficiently replicating (by Fountain codes) and sprinkling data all over, so that high availability and load resilience can be guaranteed. To support the design, *e*-SAFE also has optimization gears that enhance its performance: (*i*) Task parallelization over multiple file segments that can take advantage of an parallel processing hardwares such as an SMP, (*ii*) A *background* dissemination mechanism, that exploits lazy intervals between I/O bursts to disburse replicated information, hiding the cost of replication and dissemination from the user.

The rest of the paper is organized as follows. In section 2, we present the motivation behind our work and the challenges we face. The next section presents the overview of *e*-SAFE and the rationale behind its design. In section 4 we describe the main architectural components of *e*-SAFE followed by a short discussion on its implementation in section 5. Section 6 presents the evaluation of *e*-SAFE . We discuss the related work in section 7 and finally conclude in section 8.

## 2 Motivation and Challenges

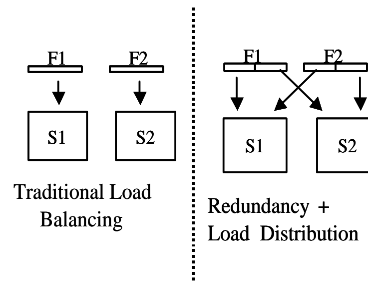



Figure 1: Load balancing vs. Distribution: S1 and S2 are *set* of servers. When most servers in S1 go down, file 1 becomes completely unavailable. However, if both files are split across S1 and S2, absence of no single set can make the files unavailable.

### Load, Availability and Distribution

Large scale data servers typically serve hundreds of clients simultaneously; thus, although such systems enjoy abundance of disk-space, the units have to serve under very heavy work-loads at recurring intervals, which calls for revisiting availability of data under high load conditions. The higher the load on a server, longer is the response time. For a client issuing a `read` request therefore, data becomes practically unavailable if it is not retrieved within a threshold latency period. Thus, a high work-load condition, which is often the case in data intensive application domains, resembles a *low availability* scenario. Standard load balancing strategies, when applied to data storage would skew the data distribution in an unfavorable way. Figure 1 shows how load balancing can skew the data distribution. Hence intuitively stretching the files across servers seem to be a better alternative. If the degree of distribution is high with a high stretch factor [1], then even at the face of a very high work-load that renders most of the servers *unavailable*, a client can read data with a reasonably low latency. In section 3, we shall discuss quantitatively the effect of redundancy on the load balancing.

### Hazards of Fragmentation

It is well understood that fragmenting data over a multitude of servers provide higher availability [43, 5, 7]. Under a given error rate it can be shown that the lifetime of data (the time after which retrieval becomes impossible with high probability due to errors creeping in the storage

[1] stretch factor $s$ is defined as the factor of redundancy, *i.e.,* , a file of size $f$, after adding redundancy bytes $s.f$ bytes long.

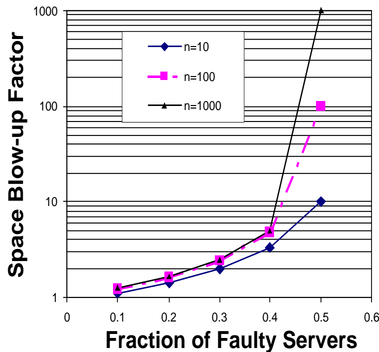



Figure 2: Space blow-up of a document fragmented over $n$ units as a function of fraction of faulty servers. The Y-axis is plotted on a logarithmic scale. Space blow-up for an optimal error correcting code is $\frac{n}{n-2t+1}$, for tolerating up to $t$ possible malicious alterations.

systems) increases rapidly with the number of encoded fragments. However, modern systems are constantly beset with security threats from agents that not only cause fail-stop behavior, but can maliciously alter the information which may go undetected unless otherwise safeguarded. Corruption of documents can also be the result of software faults; for example when the hardware or operating systems are upgraded in massively distributed systems, small patches of invisible incompatibility factors (such as a deprecated driver) may introduce wrong reads or writes resulting in *malicious like* faults. Such experiences have been reported in recent literature such as the Google File System [15].

A standard way to safeguard against such malicious failures is to attach a fingerprint vector [34, 21, 20]. Alon *et al.* observed how such schemes can result into prohibitive space blow up under a very high failure rate [2]. Intuitively, as the fragmentation increases, for a given file, the fragments become shorter while the fingerprint-vector grows larger, and thus verification information starts dominating over the actual information. Figure 2 shows how the blow-up factor grows at a high fault rate. The space-blow-up factor is plotted on Y-axis in a logarithmic scale. As the number of fragments $n$ increases the blow up increases drastically; similarly the blow up sharply jumps up as the failure rate increases for any given $n$. For example, if a file of size 100 KB is split across 100 servers designed to tolerate $t = 50$ faults using some standard optimal erasure handling techniques such as Reed-Solomon codes, one can derive that for every 2 KB of original information, there will be 2KB of additional verification information [2]. These numbers get unfavorably biased towards this overhead as $n$ grows larger. It may be argued that this blow up is only limited to the fingerprints, which is independent of the file size. However, it is often not possible to process an entire file in memory. Thus a file need to be treated in multiple segments large enough to fit in the main memory resulting in similar overhead for every segment of the file.

The designer of a large scale storage system thus confronts the following tension: On one hand, the continuous growth of hardware infrastructures, the understanding of the probability of *availability* and the possibility of parallelizing data processing and dissemination, all hint at fragmenting data over as many servers. On the other hand, the associated performance cost of reaching out for too many pieces and the resulting growth in verification information hint at limiting the number of encoded pieces. *e*-SAFE is designed to strike a balance; while *e*-SAFE can *effi ciently* take advantage of as many storage units as possible in a seamless manner by the use of modern rate-less erasure codes, at the same time, the overhead of verification information is limited to a logarithmic blow-up factor by novel use of a data-structure known as the Merkle-Tree.

# 3 System Overview

## 3.1 Design Overview

To the user *e*-SAFE offers a file system interface just like NFS. Underneath is a distributed storage system. Figure 3 depicts the broad design of *e*-SAFE . A small set of directory server serves as the meta-data server for the documents. We assume this set is highly secure and reliable and data stored in here is modifiable only by authorized access. Since this is a very small set, we believe these assumptions are not very restrictive from security management point of view. Underneath the directory server, a host of machines constitute the distributed block store. These machines can be distributed, from the span of a single building to the scale of geographically scattred. They are potentially fault prone. While locating a file, a file-system user locates the path of teh file from the directory server, which provides with the file-metada (*viz.,* Inode) that contains information about the distribution of the actual data blocks over the block-store.

Documents are encoded using a class of rate-less erasure codes (called the Fountain codes). The output of the encoding, a sequence of small blocks, is sprinkled across multiple storage units. We discuss in section 3.2 why it is important for us to have a large and flexible stretch factor. The idea is that even if some of the pieces are corrupted, there is no need to spend any maintenance effort for recovery purposes.

Documents stored in *e*-SAFE are not immutable, however, it is optimized for a class of access that mostly appends blocks to files. It is possible to edit a document

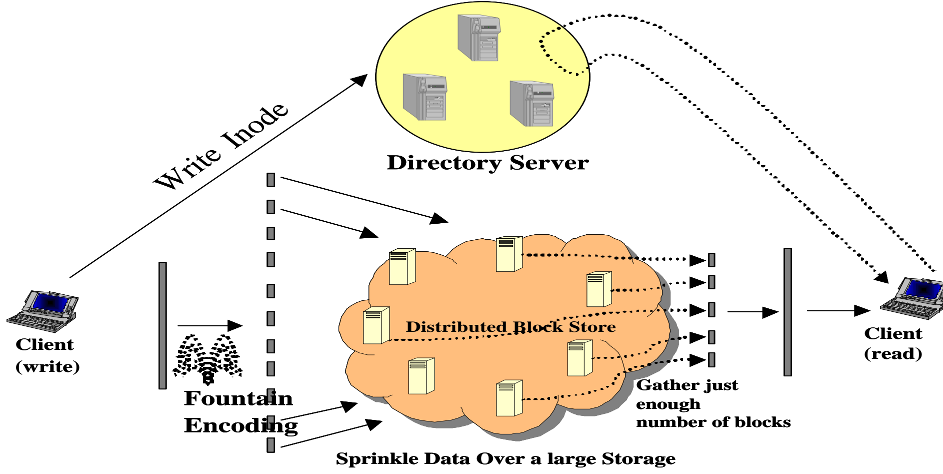



Figure 3: Overview of *e*-SAFE

from the middle as well. We assume that the meta-data information for every document (such as an *Inode*) is stored in a machine that is highly secure, *i.e.,* unauthorized modifications are not possible. We kept document encryption and key management outside *e*-SAFE . However, any traditional encryption mechanism can be integrated within the structure of *e*-SAFE without any compatibility issues. While we decided to keep the confidentiality issue outside our design goal, checks for data integrity are quite stringent.

## 3.2 Design Rationale

We investigate the question: *What does a host availability mean in the context of a distributed set of servers?* In a P2P setting, availability may simply mean the percentage of time a server is up. However, in a non-transient context, this definition is not appropriate. A good way to express the availability of a server is through the load. The availability index can be expressed as:

$$Availability\ \ Index = \frac{real\ \ response\ \ time}{optimal\ \ response\ \ time}$$

Once the above index is less than a threshold $Av_{min}$ the system may be considered *unavailable* from a client's point view. Thus the availability ($\mu$) can also be expressed as the probability :

$$\mu = Prob(Availability\ \ Index < Av_{min})$$

*e*-SAFE is targeted for very large files for which redundancy using replication becomes very expensive. As already discussed in section 1, we used Fountain codes. The great property that such codes offer is a complete flexibility of how much a file can be stretched. Suppose a file segment is coded in the $n : k$ ratio, i.e., $k$ blocks are encoded into $n$ blocks, then, ideally one could reconstruct the original segment from any $k$ blocks. Thus, given an average availability $\mu$ for each block, the net probability that the segment can be retrieved is equal to the probability that any $k$ or more blocks be available. Thus the expression is straightforward:

$$P(av) = \sum_{j=k+1}^{n} \binom{n}{j} \mu^j (1-\mu)^{(n-j)}$$

Simplifying the above equation, Bhagwan *et al.* derived a direct expression for the stretch factor $c = \frac{n}{k}$ [6],

$$c = \left( \frac{\lambda\sqrt{\frac{\mu(1-\mu)}{k}} + \sqrt{\frac{\lambda^2\mu(1-\mu)}{k} + 4\mu}}{2\mu} \right)^2$$

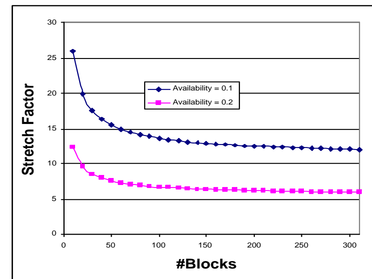



Figure 4: Stretch Factor as a function of number of initial blocks ($k$)

Figure 4 shows how the required stretch factor varies as a function of $k$, the number of pre-encoding blocks, keeping the availability ($\mu$) constant. We show two cases for

$\mu = 0.1$ and $\mu = 0.2$. It turns out that there is a sharp decrease in required $c$, as $k$ goes beyond a particular threshold. And after the sharp fall, the curves flatten out, as if increasing pre-encoding fragments does not entail further stretching. In this paper, we consider high load situations; we aren't particularly interested in high host probability. Thus we consider cases with $\mu \leq 0.1$, which means that on the average each server (or storage unit) is so loaded as to respond at most one out of 10 times, within the acceptable latency period (specified by $AV_{min}$).

So we have two points to explore in the design space. First, use a high $k$ and use a smaller stretch factor. The advantage is space-efficiency, while the drawback is high fragmentation with additional meta data overhead. Second, use a smaller number of pre-encoding blocks ($k$) and stretch further so as to cover low-availability cases. This approach is less space-efficient, but has the following advantages: (*i*) Since during a read operation data must be constructed from at least $k$ fragments, it is usually more efficient to keep this number low and (*ii*) System can handle write operations in a more time-distributed fashion; first it writes a small number of blocks just enough to reconstruct data, and then when the write burst gets over, lazily disseminate other redundancy blocks onto the persistent storage. Thus the user programs do not see high write-latency.

Keeping in mind the two factors, we designed *e*-SAFE to automatically decide the most suitable strategy within a set of constraints. Three key variables play critical roles in the choice of parameters: Length of a segment ($L_s$), the number of post-encoding blocks ($n_b$) and the size of meta-data ($L_{meta_data}$) that needs to be appended to each data block. *e*-SAFE tries to find the best parameters within a given set of constraints, such as the maximum number of servers allowed, maximum meta-data overhead allowed and so on. For a single segment that is disseminated with a stretch factor $c$ (resulting size $c.L_s$) and over $n_b$ blocks, *e*-SAFE chooses the variable parameters so the meta-data overhead is less than a given threshold $f_{overhead}$:

$$\frac{n_b.L_{meta_data}}{c.L_s} \leq f_{overhead}$$

subject to the constraints: (*i*) $N_{max} \geq n_b \geq N_{min}$ and (*ii*) $c \leq c_{max}$.

# 4 Architecture

Figure 5 gives a high level view of *e*-SAFE architecture. The top layer is a *file-system client* that provides `read`, write interfaces to the user. The next layer, FTM (Fault Tolerance Module) gathers/sends data stream to and from the FS-Client. FTM performs erasure encoding and decoding operation on the data streams. The key idea that FTM uses is that of a *Fountain Code*. We describe it shortly in section 4.1. The output of FTM is a series of data blocks. The next stage, *viz.*, the Verification layer, prepares fingerprint information for the data blocks that it receives as input. The fingerprint is generated by one way cryptographic hash function such as SHA-1. To survive malicious faults, *i.e.*, , alteration of data, or forging of identity of storage units, fingerprint of individual blocks are also then linked together into a data structure called the Merkle Tree. This is describe in section 4.2. Finally, once the data blocks are appended with appropriate fingerprint information, they are handed down to the data dissemination/aggregation layer. This stage of the software decides where to locate the server (or the storage unit) for storing or retrieving data. Presently, the search is done by computing the hash of the block-content and then indexing into a distributed hash-table. Once this decision is taken, a request followed by the payload is sent to the respective File-system servers. The FS servers is a simpler structure. On one hand it implements an RPC-based messaging protocol with the dissemination layer in the client. On the other hand this layer maintain a database of blocks indexed by their hash-values. Thus it can store or retrieve a block for the client as requested.

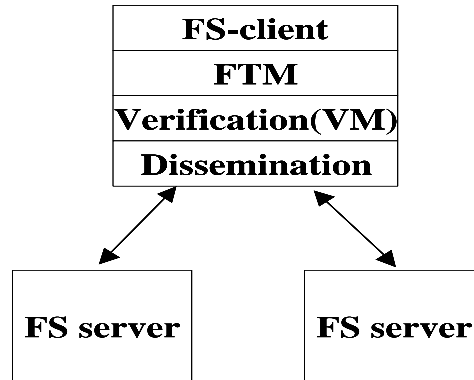



Figure 5: Broad Architecture of *e*-SAFE

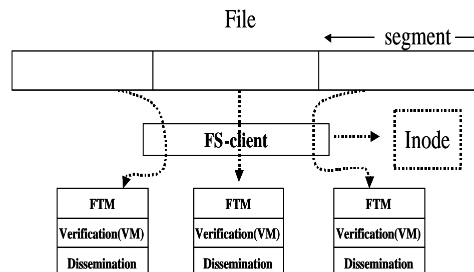



Figure 6: Parallel processing of File-Segment in *e*-SAFE

Figure 6 shows the optimization gear inherent in *e*-SAFE architecture. A large file is divided into multiple segments. Each segment is passed asynchronously to the FS-client. FS-client in terms calls the subsequent modules. The process is parallelized over different segments. Concurrent invocation of modules overlap computation/communication of multiple blocks.

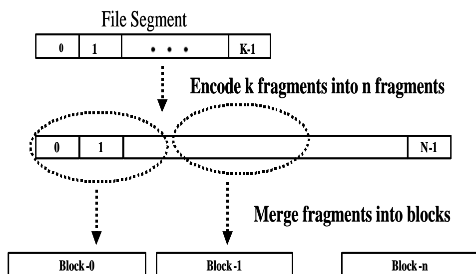



Figure 7: Preparing blocks for a File Segment *e*-SAFE

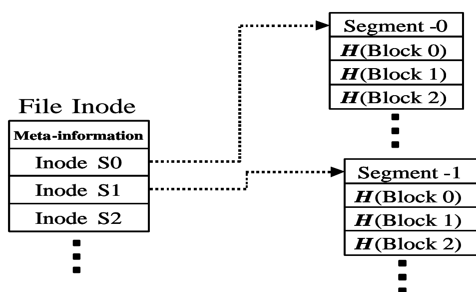



Figure 8: Structure of Inode for a multi-segment file in *e*-SAFE

Figure 7shows the flow of events on a file segment. The file segment is divided into $K$ blocks and then expanded into $N$ blocks using erasure codes. The ratio $N/K$ is the stretch factor of this code. Usually it is more efficient (and guarantees higher accuracy for probabilistic codes) to choose large $K$ and $N$. Out of the $N$ small fragments, we coalesce $N/n$ of fragments together to obtain $n$ new blocks that are independently handed down to the block-stores underneath. Note that coalescing does not change the fault tolerance limits of the system, *i.e.,* if the encoding can tolerate $f.N$ failures out of $N$ ( $f < 1$), then after coalescing at most $f.n$ can get faulty out of $n$ blocks. The scaling property inherent in the erasure codes help us adjust the scale of the system quite easily.

To support such parallel dissemination of segments, we have to maintain separate meta-data structures for each segment. Figure 8 shows how this is organized. Each file has one master Inode. The master maintains pointers to the indirect Inodes corresponding to each file segment. A segment Inode contains the content-address (typically the hash ) of the various blocks within this segment, and other other information such as number of blocks, few key encoding parameters and so on. Retrieval of Data blocks is simple - the content hash is obtained from the inode and then the main block is restored from the block-store underneath. However, if the block-store underneath is not content-addressable, then an explicitly mapping needs to be maintained.

When the number of blocks ($n$) is large, it becomes inefficient to keep the user waiting in a blocking mode till all the blocks are stored. Thus the control returns to the user only after a subset of blocks are stored ( a subset just large enough to reconstruct the segment). The write occurs in burst. Thus, the client maintains a queue of blocks in the machine's `\temp` directory and in the gap between write operations, unobtrusively and asynchronously disseminate rest of the blocks. This way one can provision for very high stretch factors and thus for very high load factors.

*e*-SAFE provides two modes of file store privilege to the user: *Permanent store with versions* and *modifi eable*. For the former, files are never removed, rather each version is maintained, with a version number. Typically the most recent version is returned, however, any previous version can also be retrieved. Maintaining consistency is trivial. In the latter case, a client, with appropriate permission, can modify existing files. In this case, the modification is done at the segment level. First, the directory server maintains consistency by serializing all write operations on a file, *i.e.,* , when a file is being modified by one user, no other user can modify it. A file modification essentially means generating new segments and deleting the old ones. The inodes at the directory servers are updated with the pointers for the new segments and similarly the segment inodes with the address of the new blocks. For the old blocks, requests are sent to the appropriate nodes, so that they can reclaim the disk-space occupied by the orphaned blocks. However, complete reclamation of these disk blocks cannot be guaranteed by *e*-SAFE , since we don't assume the nodes to be non-malicious all the time; rather accept that faults (including malicious ones) are a fact of life.

## 4.1 Erasure Coding

An erasure code works in the following way: A document is partitioned into $k$ blocks. These are called the *message blocks*. Next $n$ new blocks ($n > k$) are generated from them by adding some redundancy mechanisms, such as addition of extra parity bits and so on. The new blocks are called *encoded blocks.* Later on, the original document can be constructed from any $k$ out of $n$ encoded blocks. There are many ways erasure codes can

be generated. A very standard way is the use of the Reed-Solomon codes [33]. However, these codes are inflexible in the sense that the parameters $n$ and $k$ are static and cannot be changed on the fly. The decoding time for such codes are $\mathcal{O}(n^2)$, which means for large $n$ they become impractically slow. Moreover, these codes operate over finite field. Once a field size ($q$) is chosen, it is not possible to change it around. The maximum length of one symbol that can be considered as one unit of encoding is limited to $\log q$. Which puts a limit on the width of the stripes necessary to distribute data blocks amongst multiple storage units.

To overcome all of the above problems we used a modern class of erasure code called the *Fountain Codes* [9]. The specific version that we used is known as the LT (Luby Transform) codes [25]. LT codes are rate-less, in the sense that the stretch factor $n/k$ can be varied on-the-fly. Plus there is no limit on the symbol size - thus no restriction is imposed on the striping. And finally, these codes are inexpensive to implement and very fast in encoding (linear) and decoding times ($\mathcal{O}(n \log n)$). The category-name *fountain* is suggestive - when one needs to fill up a cup of water from a fountain, there is no need to bother about which particular droplets are being collected, rather just enough number of drops to fill in the glass would be sufficient. Rate-less codes can produce a *fountain* of encoded blocks from $k$ original message blocks. For a pre-decided small number $\epsilon$, only $(1+\epsilon)$ number of data blocks out of this fountain will suffice to reconstruct the original document. Our idea is to collect the blocks and sprinkle them over to as many storage units as necessary and thereby secure a very high *durability guarantee*.

## 4.2 Verification

We mentioned in the introduction that as data gets fragmented over more and more servers there is an impractical space overhead incurred by the verification information; especially in the face of high fraction of failure. To over come this problem we took help of a data structure called the *Merkle Tree* (MT). An MT is simply a tree (assume binary tree for the time being) while each node $j$ contains a hash value $V(j)$. The value is obtained from the child nodes; suppose $L_j$ and $R_j$ are the left and right children of the node $j$, then $V(j) = H(L_j.R_j)$, where $H()$ stands for a one-way cryptographic hash such as SHA-1 and the *dot* denotes concatenation operation.

We use the MT structure in our system in the following way. Suppose a file (or one segment of a file) is being disseminated amongst $n$ storage units. Let's assume that $n = 2^l$ for some integer $l$. Now consider an MT of depth $\log n$ and $n$ leaf nodes such that each server can be mapped to one leaf node. The hash value associated with the leaf node is the hash of the data block being sent to

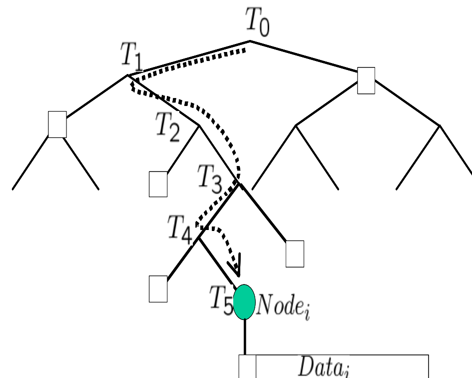



Figure 9: Construction of Verification Information from Merkle Tree: Each server is located at a leaf node. Consider server $S_i$, $T_0, T_1 \ldots$ represent the nodes on the path from the root to the $i$-th server. At any internal node of the tree the hash value $V$ is prepared as $V(j) = Hash(V(left_chid).V(right_chid))$. The final verification information sent to $S_i$ is the collection of hash values stored at the siblings of the vertices on this path.

the corresponding server. Once the leaf hash-values are directly obtained from the corresponding data blocks, the intermediate levels of hashes are easily computed all the way up to the root. Figure 9 depicts this MT. Now, consider the $i$-th server ( alternatively, the $i$-th leaf node), and the unique path from the root to this node. The vertices on this path are $T_0, T_1$ and so on. We collect the hash values associated with the vertices *that are siblings* of $T_1 \ldots T_5$. Since $T_0$ has no sibling and we keep this value separately as the Root-Hash. In the figure all the siblings are marked by a 'square'; their values are collected and packed into the verification information sent to the $i$-th server. While retrieving a block, one needs to re-compute the hash and then successively recompute the hashes up its path to the root (with the help of the sibling hash-values stored) and then finally verify if it is matching with the root-hash. If this test is passed, the data block's integrity can be considered intact with very high probability.

# 5 Implementation

*e*-SAFE is currently implemented as a user level library written in C++ and has been tested to run on Linux. We used SHA-1 for hashing purposes, for verifying data blocks as well as creating block identifiers. The data dissemination layer currently has two implementations. The first one uses DHash [**?**]. DHash is built on top of a scalable P2P look-up system called Chord [39]. In our DHash based implementation of *e*-SAFE , each inode maintains a list of its block identifiers. During a store or fetch op-

eration, lookup is initiated into the Chord network. Once a machine with appropriate nodeID is obtained, the data block is directly exchanged with it. The Inode server does not need to exclusively maintain the mapping between the blocks of a file and the servers they are stored in; that is left to DHash. Although Chord was designed to cater to P2P like systems, it has been later used to build wider area file system. Moreover, DHash performance has been enhanced (by a factor of two) by use of a new transport scheme called the Striped Transport Protocol(STP). Thus we chose to use the STP based DHash implementation for our purpose.

We also implemented another version of the block store that does not depend on DHash. In this implementation, the Inode server maintains the mapping between blocks and the servers they are stored in. We shall refer to this implementation as the Simple Block Store (SBS) version. Currently, the blocks are randomly assigned to servers from a list of server hosts, and this information has to be explicitly maintained at the Inode server. This way of choosing servers runs the risk of skewed load distribution (which is one of the reasons why we first chose DHash as our implementation vehicle). For SBS, once the client obtains location information for a block, it contacts the respective server through an RPC interface. The end-to-end performance of this implementation is better than DHash, because the chord-based lookups are replaced by a direct client-server contact. SBS has two versions, synchronous and asynchronous. The former is based completely upon the synchronous RPC implementation available on Linux. The latter is an extension to it. In the asynchronous version, the RPC client maintains an extra thread receiving any incoming packets from the server asynchronously. Similarly, on the server side, an asynchronous service thread is maintained, along with a request queue. The standard RPC request handler enqueues requests which are asynchronously serviced by the service thread.

# 6 Evaluation

we evaluated *e*-SAFE both on a series of microbenchmarks and as well as under workloads. The objective of the microbenchmarking was two fold: (*i*) quantify at a micro level how much the read and write operation costs over different implementations the block store, and (*ii*) tease out the total times taken to inspect how different layers contribute to the total cost of opearations.

All our experiments were performed on a set of fourteen dual processor SMP machines, 2.8 GHz Intel Xeon processor, with 2 GB SDRAM and 512 KB L2 cache. They are connected by a switched Gigabit Ethernet. In all the following experiments, the machines were partitioned into two physical sets - servers, hosting the block-store (acting as *e*-SAFE FS-server, ref. Figure 5), and the clients that are outside the ring of servers, and only generate store and fetch operation for the servers. The client machines do not store anything. This division was effected partly because we wanted to separate the performance interferences of clients and servers and also to mimic the structure of a separate storage subsystem catering to its clients.

## 6.1 Microbenchmarks

We primarily study the latency taken by *e*-SAFE for storing and fetching files of different sizes. The latency increases as we split files across more and more servers, *i.e.,* split files into more and more blocks. *e*-SAFE is limited in performance by the bandwidth offered by the block-store underneath. Thus we examined *e*-SAFE under two different implementations of the block-store that we described in section 5.

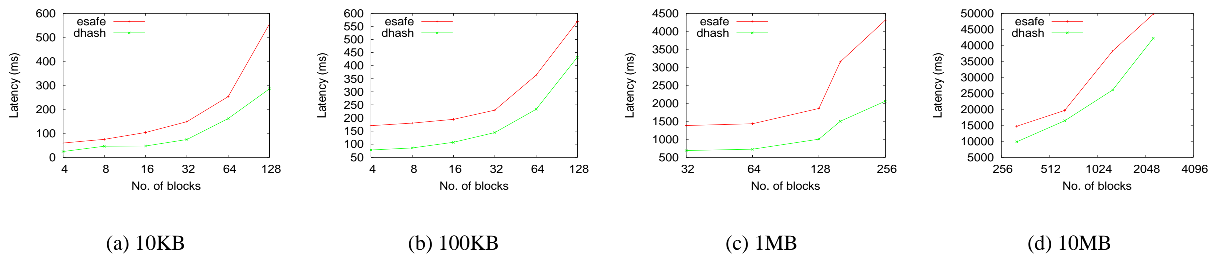
## Performance of *e*-SAFE on DHT

### End-to-end Latency

The first experiment that we performed is the following. An *e*-SAFE client writes a file of size $s$ onto the store, split into $b$ blocks. The latency is recorded. To compare with the above operation, the same file, split into equal number of blocks, is handed over to the raw DHash layer for storage. Since *e*-SAFE performs additional operations, such as stretching up the files by encoding operation, processing and adding verification information, and appending meta-data that is needed to support subsequent decode and verify operations, the expectation is that *e*-SAFE would perform worse compared to the raw DHash layer. Figure 10 shows the comparisons for different file sizes. On X-axis, we measured the number of blocks (essentially the number of different servers the file was distributed to) and Y-axis measure the latency. We show the comparative latencies across various file sizes. From the latency values we see that *e*-SAFE follows quite closely the performance of DHash . These numbers show that the top three layers of *e*-SAFE do not add significantly to the overall latency.

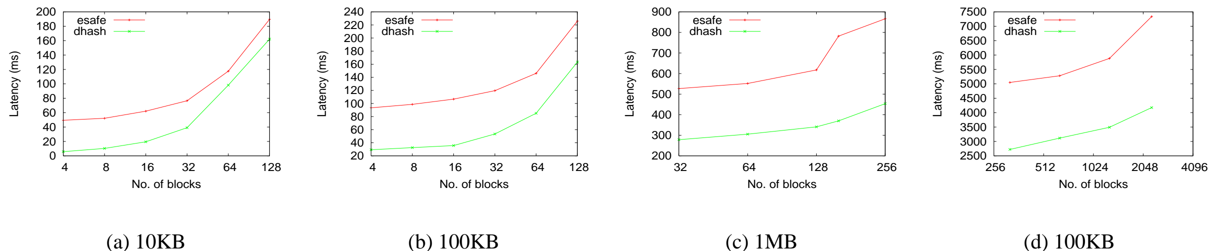
Figure 11 describes the results of similar experiments for the read operations. In this case, *e*-SAFE latency follow the baseline DHash latency, however, as the data size increases, we see that this gap widens. *e*-SAFE still performs quite well though; for example splitting a 10MB file onto approximately 300 blocks, yields a latency of less than 3 seconds and thus yielding a throughput of over 3MB/s.

Writes are more expensive than reads. During a write operation, the scheme produces many more blocks than




(a) 10KB (b) 100KB (c) 1MB (d) 10MB

Figure 10: Performance of *e*-SAFE over Raw DHash delivery




(a) 10KB (b) 100KB (c) 1MB (d) 100KB

Figure 11: Performance of *e*-SAFE over Raw DHash delivery

the original document has. This is decided by the stretch factor. In above experiments, we used a stretch factor of 2.0. However, higher latencies are expected (and observed) as we increase this factor. Reads however are somewhat independent of the stretch factor. A read succeeds as soon as enough number of blocks are gathered. Plus, caching in the DHash layer facilitates the reads by reducing the look-ups.

### Dissection of Cost

Figure 12 gives a split of the latency (write operation) into two parts: the time spent in the top three layers, and time spent in the dissemination layer. The numbers clearly indicate that additional computations done by *e*-SAFE is quite minimal compared to the time spent in networking.

We repeat the similar set of experiments for file reads. Figure shows these performance figures. Again the read latencies for *e*-SAFE and DHash match quite closely. Figure 13 tease out the time for read operations. Here we see that network times are substantially reduced compared to the computation. This has happened for two reasons. First, as we already pointed out, it is not necessary to collect all the data blocks that were written. In addition, the computational overhead for a read is much more than a write. This is because the decoding operation is almost twice as expensive as encoding, verifying a block is slightly faster than generating verification for the entire file.

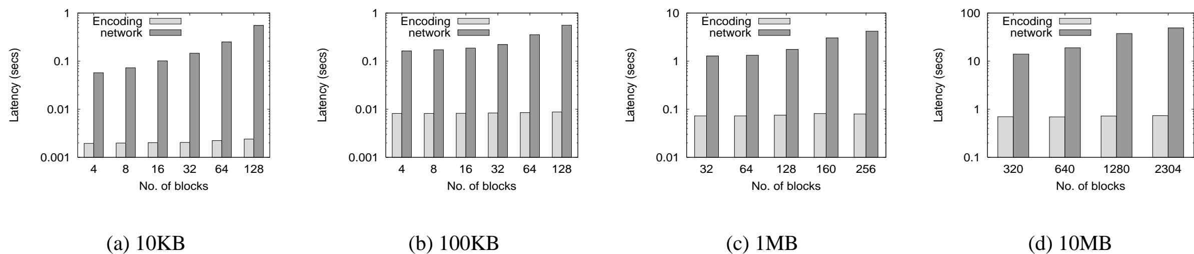
### Performance of the SBS version

We performed similar experiments on the other version of data dissemination, *viz.,* the SBS implementation. Since this version explicitly maintains a mapping between block ids and servers, no look-up is necessary. Figure 14 presents the numbers for the write operations (in a similar experimental setting as before). As before, the X-axis denotes the number of blocks and the Y-axis denotes the latency. We observe that SBS writes are much faster compared to DHash. As we already pointed out, SBS does not need any lookup to find out an appropriate server, since the block to server mapping is readily available in the Inode server. We see this across all the data sizes (only a small representative set is presented here). The performance of the SBS version is quite encouraging; a file of size of 10MB is written in roughly 2 seconds yielding a bandwidth of 5MB/s.

Figure 15 shows the performance of `read` operation. And we see extremely fast reads happening in this setting. This supports our belief that *e*-SAFE has a suitable architecture for high performance operations.

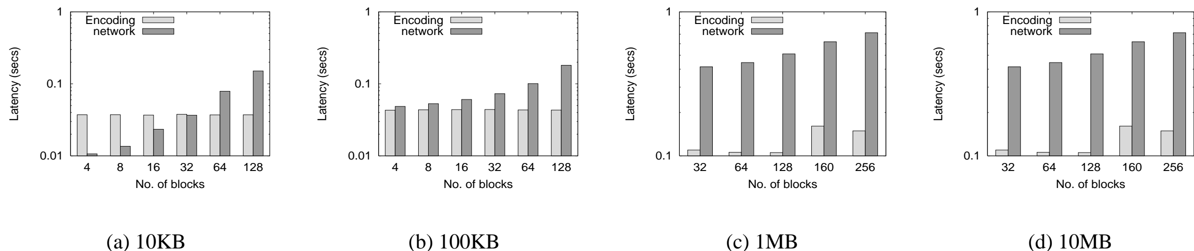
## 6.2 Performance under Workload

Next we tested *e*-SAFE under workload. Unfortunately we did not have real I/O traces for such a system available to us. One choice would be to run few realistic applications. Again, since such a storage environment could be




(a) 10KB (b) 100KB (c) 1MB (d) 10MB

Figure 12: Split of time in Computation and Networking(Write)




(a) 10KB (b) 100KB (c) 1MB (d) 10MB

Figure 13: Split of time in Computation and Networking (Read)

catering needs of diverse situation, it is more important that we test it on work-loads of very generic nature. Our purpose therefore, is to generate synthetic workloads that approximates the characteristic of many different work-loads.

I/O workload characterization has received wide attention in the past, and experiences reveal that typical read write requests that occur over various domains and scales (such as disks, network, web) follow some well defined patterns [17, 24, 13, 42]. First, I/O is bursty - both read and write requests appear in short bursts with intermediate lean periods. Second, I/O traffic bears strong self-similarity, *i.e.,* if one zooms into smaller and smaller intervals of an extended I/O trace, the patterns over smaller intervals resemble those over longer intervals. Third, there is a strong resemblance to these traces with the 80-20 rule often observed in Database systems [19] This rule roughly says approximately 80% of query traffic experienced by a machine queries 20% of data. To model such behaviors, we used a trace generation model called the *bmodel* [42]. It has been shown that this model accurately approximates I/O behaviors of various different systems. The model is dependent on a parameter $b$, called the bias. A bias of 0.8 corresponds to the factor 80% in the 80/20 law. *bmodel* also generates self-similar traffic. Self similarity in traces is usually measured by an index $H$, known as the *Hurst Exponent* [17]. The Hurst exponent of a trace generated by a bias $b$ is given by: $H \approx \frac{1}{2} - \frac{1}{2}(b^2 + (1-b)^2)$. Thus using bmodel allowed us to vary the characteristic of the traces to create family of work-loads that are fairly generic and representative in nature.

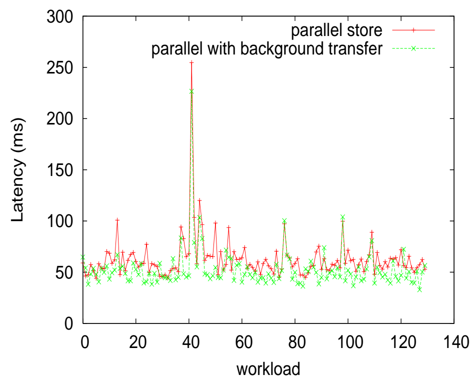
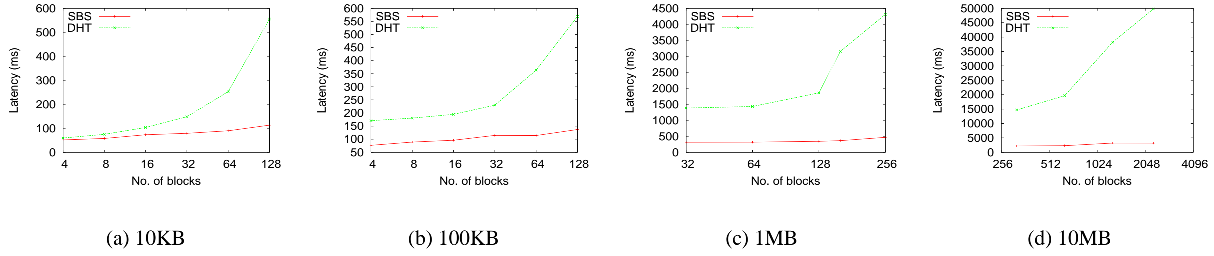


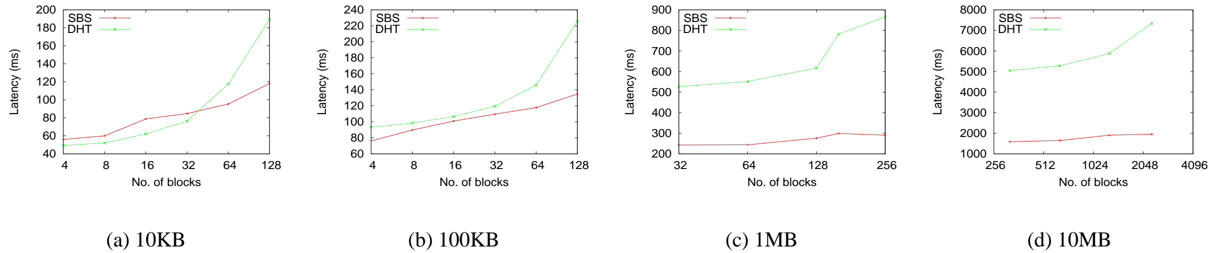

Figure 16: Latency distribution over a workload

We carried out the following simple experiments with the workloads. Synthetic I/O traces were generated for both reads and writes for a given time interval. Next we initiated a client to write/read by following those traces. Figure 16 presents a scenario of this workload for write operation. A total of 100 MB of data was distributed to be written over a period of 200 seconds. The distribution (into different chunk sizes as produced by the *bmodel*, is spaced evenly over the 200 sec interval. In Figure 16, X-axis denotes the entries of the workload (for various data




(a) 10KB (b) 100KB (c) 1MB (d) 10MB

Figure 14: Write latencies of DHash and SBS implementations of *e*-SAFE




(a) 10KB (b) 100KB (c) 1MB (d) 10MB

Figure 15: Read latencies of DHash and SBS implementations of *e*-SAFE

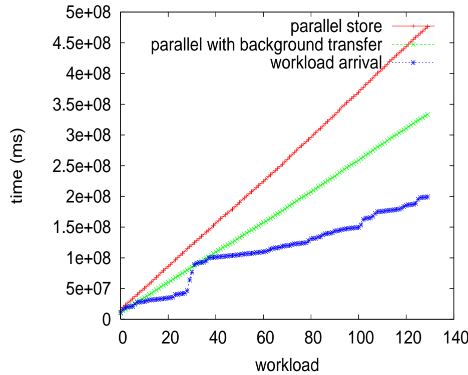



Figure 17: Back-logs generated by the workload

sizes), in the order they were generated, and on Y axis, we measure the latency they encountered for the write to finish. Since they arrive in that order in time, the X-axis can be also be treated as a time axis (for arrival of write requests). We latencies for two different modes of storing; background dissemination on and off. The stretch factor in these experiments were kept at 2.0. As expected, latter mode results into higher latency. However, as the I/O burst continue, the write requests would queue up and thus building on a backlog. The system cleans up this backlog during the lean phases of I/O, which comes immediately following this burst.

In Figure, 17, we show how the backlog builds up. The X-axis denotes the requests arriving in that order. The Y axis measures time. Thus, the first line represents the precise arrival time of requests relative to work-load window. The next line shows the time these request calls returned with minimal storing, *i.e.,* , storing just enough for reconstruction, and leaving the rest for the background dissemination. The third line in this figure describe the service time of the requests if data dissemination happened all in foreground. Clearly, the background process does reduce the backlog, however, it does introduce partial backlogs, *i.e.,* parts of incomplete writes. For reads that happen long time after the writes, the background dissemination would clear up this partial backlog. The large size of local disks help us contain these backlogs. However, for reads that are too closely spaced with the write, it is fair to assume that the nodes on which data (just enough for reconstruction) got written, are still available and thus should be able to supply the necessary blocks.

Finally, we wanted to demonstrate the effect of higher stretch factor that we outlined in earlier section. For this experiment, we use the asynchronous version of the SBS. For the experiments reported here, we used a file of size 240 KB. We stored this file using stretch factor 2 and 4. In the former case, we split the file into 16 blocks, and in the latter 64 blocks of the same size as in the former case. Now we performed `fetch` operation on the file assuming the servers are highly loaded. We simulated a loaded server in the following way; for returning every

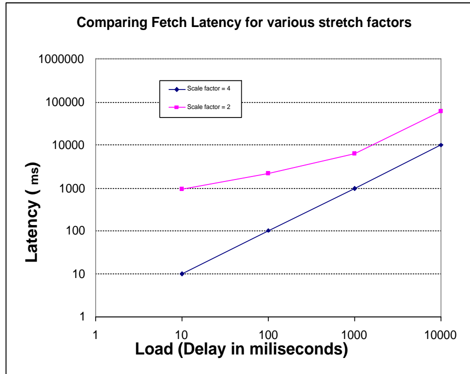



Figure 18: Back-logs generated by the workload

block a random delay was inserted between 0 and $\delta$ time units. A higher value of $\delta$ would emulate higher load on the servers. The two results are plotted in Figure 18. $\delta$ was varied from 10ms to 10 seconds. As we have reasoned before, the response for the case of stretch factor 4 is much better than the case of stretch factor 2; it is literally ten to hundred times better, which makes the case for using high stretch factor and thus getting high availability.

# 7 Related Work

Distributed Storage is an age-old concept that has received wide attention in multiple contexts. Thus, a long list of related literature predates our work.

## Distributed Storage Systems

Distributed storage hardware such as RAID [11] existed and continue to be improved in the current practice. Systems such as Petal [23] exported through virtualization a distributed set of disks seamlessly to clients. For large scale organizations, having a multitude of storage units, the natural evolution was to have systems such as a Storage Area Network (SAN) or network attached storage (NAS) [31]. Modern systems are emerging out to the scale of the internet such as the Internet Protocol Storage Area Network (IPSAN) [32]. Clearly, such growth in scale, and the fall in price of storage hardware complements our work.

Close to our work is the area of distributed file systems. Significant research has been done in this space since the Andrew File system [**?**]. A few more recent examples are xFS [4], Frangipani (on top of Petal) [40], and so on. With the growth of the internet, the focus shifted on to realizing file systems over wider areas. Security became one of the most critical issues. SFS is file system based on separation of key management from file system so that it can span the internet with heterogeneous key management policies. The key idea in SFS is the use of self-certifying path name. FARSITE system [1] enables a peer to peer like environment of mutually distrusting desktops to provide a highly available and secure file system; it takes care of making copies of replicas and metadata when desktops leave systems. Security involves two aspects, *viz.,* , confidentiality and integrity. While, confidentiality is not a big problem in systems that are centrally maintained (regardless of being distributed), data integrity is critical, because faults, both in the form of fail-stop and data corruption creeps in for various reasons. Wide area file systems, such as CFS, [14] make sure that integrity and availability are maintained with effective replication caching. The Google file system(GFS) deploys a distributed file system spanning literally thousands of servers that deliver high bandwidth and availability [15]. GFS provides availability by splitting files into blocks and then replicating each block along with additional checksums, very much along the line of CFS.

## P2P File systems and look-ups

In recent years P2P systems received very wide attention. While the span for P2P systems is the internet, the scale is also overwhelmingly large, *i.e.,* literally millions of servers and their dynamics need to be considered. P2P systems opened up a plethora of new possibilities, of which file sharing became popular even at a commercial level. Oceanstore [22, 35] system first attempted to harness the astronomically high amount of storage that might be reached through the internet to create an archival level persistent storage capable of enduring very long time. One of the key ingredients in its design is the use of erasure codes. Erasure codes have been shown to provide higher durability guarantee than naive replication at a much lower space cost [43]. Very recently, Bhagwan *et al.* reported TotalRecall (TR), a P2P based file system that is designed to handle the dynamic behavior of P2P systems. TR is based on the observation that in a P2P system, nodes join and disappear in a diurnal patter over a short run, and in the long run many of the nodes leave the system permanently. To handle such scale of dynamics, TR deploys an availability monitoring unit, that checks for the availability of files in a periodic basis and repairs a file back to the desired availability whenever this factor falls below a threshold. Understanding the dynamics of P2P systems has been an important ingredient of the understanding of the availability [5, 6, 8]. In has been sometimes argued that in a P2P like environment, only a very small fraction of the nodes cooperate meaningfully and on a permanent basis, while the rest disappear mostly after a short period. However, we do not assume an environment such as P2P. Our setting resembles more to a

server farm that grew out of many inexpensive local disks available individually. In our design we decided to use a high stretch factor to replace periodic repairs. However, this will not work in a P2P context, because irrespective of the stretch factor, data, will be permanently lost in the face of the constant decay [6].

### Fault tolerance

There are two approaches to handle faults: (*i*) Pure replication and (*ii*) Erasure coding. Early on, Quorum systems [3] have been used to provide coordination in distributed systems. Quorum approach is pure replication based. Early works on quorum system considered how to handle benign failures [16, 41]. Byzantine failures, where the servers maliciously corrupt data, and collude among themselves, were studied later on [26, 30, 27]. The replication techniques studied in these investigations were adopted in the design of persistent object stores, such as Phalanx [28] and Fleet [29]. Another alternative to handle byzantine faults in a distributed environment is replicated state machine approach [37]. Castro and Liskov [10] presented a practical implementation based on this approach; they built a file system that can handle byzantine faults.

Erasure Coding approaches are more space optimal. In a seminal paper, Rabin presented the first Information Dispersal Algorithm (IDA) that can be used for fault tolerance in parallel and distributed systems. IDA is essentially a kind of erasure coding. Krawczyk [20] extended the IDA scheme to handle Byzantine faults, by appending fingerprints of each data piece along with the fingerprint of the entire content. However, the distributed fingerprinting can be combined with secret sharing [38] in a clever way that uses symmetric key encryption; the resulting scheme is shown to be secure with short secret sizes [21]. This approach, known as **SecureIDA** was exploited in the design of e-Vault, an electronic storage system developed at IBM [18].

## 8 Conclusions and Future Work

We discussed the design of *e*-SAFE , a distributed storage service targeted for very large scale decentralized storage. In the design of *e*-SAFE we made a quantitative observation, that of equating high load with low availability. Based at the heart of *e*-SAFE 's design, is a special class of code, called the Fountain codes, that makes *e*-SAFE seamlessly adaptable to unlimited stretching and thus to hardware extensions of any degree. By the virtue of the same codes, *e*-SAFE can sprinkle data around to any stretch factor, and thus can reduce management overhead to a great extent. As part of the ongoing and future work, we are investigating the dynamics of workloads more closely and the resulting performance of *e*-SAFE . Diverse loads and complicated asynchronous behaviors of various components leave open a plethora of questions that can only be answered by combining more analytical studies such as random processes and queueing theory techniques with our experimental methods. Such a study is underway.

## References


[1] A. Adya and et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation(OSDI)*, 2002.

[2] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. P. Stern. Scalable secure storage when half the system is faulty. In *Automata, Languages and Programming*, pages 576–587, 2000.

[3] Y. Amir and A. Wool. Optimal availability quorum systems: Theory and practice. *Information Processing Letters*, 65(5):223–228, 1998.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Pat-terson, D. Roselli, and R. Wang. Serverless network fi le systems. In *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[5] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[6] R. Bhagwan, S. Savage, and G. M. Voelker. Replication strategies for highly available peer-to-peer storage systems. Technical Report CS2002-0726, University of California, San Diego, 2002.

[7] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total recall: System support for automated availability management. In *Proceedings of the First ACM/Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[8] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two.

[9] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM*, pages 56–67, 1998.

[10] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

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

[12] A. Chien. Computing elements. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 567–592. Morgan Kaufmann, 2004.

[13] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of SIGMETRICS'96: The ACM International Conference on Measurement and Modeling of Computer Systems.*, Philadelphia, Pennsylvania, May 1996. Also, in Performance evaluation review, May 1996, 24(1):160-169.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[15] S. Ghemawat, H. Gobioff, and S.-T. leung. The google file system. In *Proceedings of SOSP*, October 2004.

[16] D. Gifford. Weighted voting for replicated data. 1979.

[17] W. W. Hsu and A. Smith. Characteristics of i/o traffic in personal computer and workload servers. *IBM SYSTEMS JOURNAL*, 42(2), 2002.

[18] A. Iyengar, R. Cahn, J. Garay, and C. Jutla. Design and implementation of a secure distributed data repository. 1998.

[19] Jim Gray *et al.* . Quickly generating billion-record synthetic databases. In *Proceedings of SIGMOD*, 1994.

[20] H. Krawczyk. Distributed fingerprints and secure information dispersal. In *Proc. 13th ACM Symp. on Principles of Distributed Computating*, pages 207–218, 1993.

[21] H. Krawczyk. Secret sharing made short. *Advances in Cryptology ( CRYPTO)*, 773:136–146, 1994.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[23] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, 1996.

[24] W. E. Leland, M. S. Taqq, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic. In D. P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[25] M. Luby. Lt codes. In *Proceedings of 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002.

[26] D. Malkhi and M. Reiter. Byzantine quorum systems. pages 569–578, 1997.

[27] D. Malkhi, M. Reiter, and A. Wool. Optimal byzantine quorum systems. Technical Report 97-10, 17, 1997.

[28] D. Malkhi and M. K. Reiter. Secure and scalable replication in phalanx. In *Symposium on Reliable Distributed Systems*, pages 51–58, 1998.

[29] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *The 2nd DARPA Information Survivability Conference and Exposition*.

[30] D. Malkhi, M. K. Reiter, and A. Wool. The load and availability of byzantine quorum systems. In *Symposium on Principles of Distributed Computing*, pages 249–257, 1997.

[31] R. J. Morris and B. J. Truskowski. Evolution of storage area networks. *IBM SYSTEMS JOURNAL*, 42(2), 2002.

[32] Prasenjit Sarkar *et al*. Internet protocol storage area networks. *IBM SYSTEMS JOURNAL*, 42(2), 2002.

[33] O. Pretzel. *Error Correcting Codes and Finite Fields*. Clarendon Press, Oxford, 1992.

[34] M. Rabin. The efficient dispersal of information for security, load balancing, and fault tolerance. *JACM*, 36(5):335–348, April 1989.

[35] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the Conference on File and Storage Technologies*. USENIX, 2003.

[36] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[37] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.

[38] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[39] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings ACM SIGCOMM*, Aug 2001.

[40] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[41] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. In *Database Systems*, volume 4, pages 180–209, 1979.

[42] M. Wang, T. M. Madhyastha, N. H. Chan, S. Papadimitriou, and C. Faloutsos. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *ICDE*, 2002.

[43] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems: First International Workshop (IPTPS)*, 2002.

[44] Wee Teck Ng and Bruce Hillyer. Obtaining high performance for storage outsourcing. In *Proceedings SIGMETRICS/Performance*, pages 322–323, 2001.

[45] Wee Teck Ng *et al.* Obtaining high performance for storage outsourcing. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2002.