Reclaiming Space from Duplicate Files in a Serverless Distributed File System

John R. Douceur
Atul Adya
William J. Bolosky
Dan Simon
Marvin Theimer

July 2002

Technical Report
MSR-TR-2002-30

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Reclaiming Space from Duplicate Files in a Serverless Distributed File System*

John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, Marvin Theimer
Microsoft Research
{johndo, adya, bolosky, dansimon, theimer}@microsoft.com

Abstract

The Farsite distributed file system provides availability by replicating each file onto multiple desktop computers. Since this replication consumes significant storage space, it is important to reclaim used space where possible. Measurement of over 500 desktop file systems shows that nearly half of all consumed space is occupied by duplicate files. We present a mechanism to reclaim space from this incidental duplication to make it available for controlled file replication. Our mechanism includes 1) convergent encryption, which enables duplicate files to coalesced into the space of a single file, even if the files are encrypted with different users' keys, and 2) SALAD, a Self-Arranging, Lossy, Associative Database for aggregating file content and location information in a decentralized, scalable, fault-tolerant manner. Large-scale simulation experiments show that the duplicate-file coalescing system is scalable, highly effective, and fault-tolerant.

1. Introduction

This paper addresses the problems of identifying and coalescing identical files in the Farsite [8] distributed file system, for the purpose of reclaiming storage space consumed by incidentally redundant content. Farsite is a secure, scalable, serverless file system that logically functions as a centralized file server but that is physically distributed among a networked collection of desktop workstations. Since desktop machines are not always on, not centrally managed, and not physically secured, the space reclamation process must tolerate a high rate of system failure, operate without central coordination, and function in tandem with cryptographic security.

Farsite's intended purpose is to provide the advantages of a central file server (a global name space, location-transparency, reliability, availability, and security) without the attendant disadvantages (additional expense, physical plant, administration, and vulnerability to geographically localized faults). It provides high availability and reliability – while executing on a substrate of inherently unreliable machines – primarily through a high degree of replication of both file content and directory infrastructure. Since this deliberate and controlled replication causes a dramatic increase in the space consumed by the file system, it is advantageous to reclaim storage space due to incidental and erratic duplication of file content.

Since the disk space of desktop computers is mostly unused [13] and becoming less used over time [14], reclaiming disk space might not seem to be an important issue. However, Farsite (like most peer-to-peer systems [32]) relies on voluntary participation of the client machines' owners, who may be reluctant to let their machines participate in a distributed file system that substantially reduces the disk space available for their use.

Measurements [8] of 550 desktop file systems at Microsoft show that almost half of all occupied disk space can be reclaimed by eliminating duplicate files from the aggregated set of multiple users' systems. Performing this reclamation in Farsite requires solutions to four problems:

1) Enabling the identification and coalescing of identical files when these files are (for security reasons) encrypted with the keys of different users.

2) Identifying, in a decentralized, scalable, fault-tolerant manner, files that have identical content.

3) Relocating the replicas of files with identical content to a common set of storage machines.

4) Coalescing the identical files to reclaim storage space, while maintaining the semantics of separate files.

The latter two of these problems are addressed by a file-replica-relocation system [14] and the Windows® 2000 [34] Single-Instance Store (SIS) [7], which are described in other publications. In the present paper, we describe the first two problems' solutions: 1) *convergent encryption*, a cryptosystem that produces identical ciphertext files from identical plaintext files, irrespective of their encryption keys and 2) *SALAD*, a *Self-Arranging, Lossy, Associative Database* for aggregating and analyzing file content information. Collectively, these components are called the *Duplicate-File Coalescing* (DFC) subsystem of Farsite.

The following section briefly describes the Farsite system. Section 3 explains convergent encryption and formally proves its security. Section 4 describes both the steady-state operation of SALAD and the maintenance of SALAD as machines join and leave the system. Section 5 shows results from large-scale simulation experiments using file content data collected from a set of 585 desktop file systems. Section 6 discusses related work, and Section 7 concludes.

*This paper is an extended version of a paper that appeared in the 22nd IEEE International Conference on Distributed Computing Systems [15].

2. Background – the Farsite file system

Farsite [8] is a scalable, serverless, distributed file system under development at Microsoft Research. It provides logically centralized file storage that is secure, reliable, and highly available, by federating the distributed storage and communication resources of a set of not-fully-trusted client computers, such as the desktop machines of a large corporation. These machines voluntarily contribute resources to the system in exchange for the ability to store files in the collective file store. Every participating machine functions not only as a client device for its local user but also both as a *file host* – storing replicas of encrypted file content on behalf of the system – and as a member of a *directory group* – storing metadata for a portion of the file-system namespace.

Data privacy in Farsite is rooted in symmetric-key and public-key cryptography [27], and data reliability is rooted in replication. When a client writes a file, it encrypts the data using the public keys of all authorized readers of that file, and the encrypted file is replicated and distributed to a set of untrusted file hosts. The encryption prevents file hosts from unauthorized viewing of the file contents, and the replication prevents any single file host from deliberately (or accidentally) destroying a file. Typical replication factors are three or four replicas per file [8, 14].

The integrity of file content and of the system namespace is rooted in replicated state machines that communicate via a Byzantine-fault-tolerant protocol [11]. Directories are apportioned among groups of machines. The machines in each directory group jointly manage a region of the file-system namespace, and the Byzantine protocol guarantees that the directory group operates correctly as long as fewer than one third of its constituent machines fail in any arbitrary or malicious manner. In addition to maintaining directory data and file metadata, each directory group also determines which file groups store replicas of the files contained in its directories, using a distributed file-replica-placement algorithm [14].

For security reasons, machines communicate with each other over cryptographically authenticated and secured channels, which are established using public-key cryptography. Therefore, each machine has its own public/private key pair (separate from the key pairs held by users), and each machine computes a large (20-byte) unique identifier for itself from a cryptographically strong hash of its public key. Since the corresponding private key is known only by that machine, it is the only machine that can sign a certificate that validates its own identifier, making machine identifiers verifiable and unforgeable.

Each directory group needs to determine which of the files it manages have contents that are identical to other files that may be managed by another directory group. Each file host needs to be able to coalesce identical files that it stores, even if they have been encrypted separately.

3. Convergent encryption

If Farsite were to use a conventional cryptosystem to encrypt its files, then two identical files encrypted with different users' keys would have different encrypted representations, and the DFC subsystem could neither recognize that the files are identical nor coalesce the encrypted files into the space of a single file, unless it had access to the users' private keys, which would be a significant security violation. Therefore, we have developed a cryptosystem – called *convergent encryption* – that produces identical ciphertext files from identical plaintext files, irrespective of their encryption keys.

To encrypt a file using convergent encryption, a client first computes a cryptographically strong hash of the file content. The file is then encrypted using this hash value as a key. The hash value is then encrypted using the public keys of all authorized readers of the file, and these encrypted values are attached to the file as metadata. Formally, given a symmetric-key encryption function E, a public-key encryption function F, a cryptographic hash function H, and a public/private key pair (K_u, K'_u) for each user u in a set of users U_f of file f, convergently encrypted file ciphertext C_f is a ⟨data, metadata⟩ tuple given by function X applied to file plaintext P_f:

$$C_f = \mathrm{X}_{K_u}\left(P_f\right) = \left\langle c_f, \mathrm{M}_f \right\rangle \qquad (1)$$

Where the file data ciphertext c_f is the encryption of the file data plaintext, using the plaintext hash as a key:

$$c_f = E_{H(P_f)}\left(P_f\right) \qquad (2)$$

And the ciphertext metadata M_f is a set of encryptions of the plaintext hash, using the users' public keys:

$$\mathrm{M}_f = \left\{ \mu_u \ni \mu_u = F_{K_u}\left(H\left(P_f\right)\right) \wedge u \in U_f \right\} \qquad (3)$$

Any authorized reader u can decrypt the file by decrypting the hash value with the reader's private key K'_u and then decrypting the file content using the hash as the key:

$$P_f = \mathrm{X}^{-1}{}_{K'_u}\left(C_f\right) = E^{-1}{}_{F^{-1}{}_{K'_u}(\mu_u)}\left(c_f\right) \qquad (4)$$

Because the file is encrypted using its own hash as a key, the file data ciphertext c_f is fully determined by the file data plaintext P_f. Therefore, the DFC subsystem, without knowledge of users' keys, can 1) determine that two files are identical and 2) store them in the space of a single file (plus a small amount of space per user's key).

3.1. Convergent encryption – proof of security

Convergent encryption deliberately leaks a controlled amount of information, namely whether or not the plaintexts of two encrypted messages are identical. In this subsection, we prove a theorem stating that we are not accidentally leaking more information than we intend. To obtain a general result regarding our construction, not a result specific to any realization using particular encryption or hash functions, our proof is based on the random oracle model [4] of cryptographic primitives.

We are given a security parameter (key length) n and a plaintext string P of length $m \geq n$, where P is selected uniformly at random from a superpolynomial (in n) subset S of all possible strings of length m: $P \in_r S \ni \forall s\ [s \in S \rightarrow s \in \{0,1\}^m] \wedge |S| = n^{\omega(1)}$. This assumption is stronger than that allowed in a semantic security proof, since convergent encryption explicitly allows the contents of an encrypted message to be verified without access to the encryption key. According to the random oracle model, the hash function H is selected uniformly at random from the set of all mappings of strings of length m to strings of length n: $H \in_r \{h \ni h\text{: } \{0,1\}^m \rightarrow \{0,1\}^n\}$. The encryption function E is selected uniformly at random from the set of all permutations that map keys of length n and plaintext strings of length m to ciphertext strings of length m: $E \in_r \{e \ni e\text{: } \{0,1\}^n \times \{0,1\}^m \rightarrow \{0,1\}^m \wedge \forall k,p_1,p_2\ [p_1 \neq p_2 \rightarrow e_k(p_1) \neq e_k(p_2)]\}$. Furthermore, since the encryption function is symmetric, there exists a function E^{-1} that is the inverse of function E given k: $\forall k,p\ [E^{-1}_k(E_k(p)) = p]$. (This is not an inverse function in the traditional sense, since it yields only the plaintext and not the encryption key.)

We assume that H, E, and E^{-1} are all available to an attacker. However, since these functions are random oracles, the only method by which the attacker can extract information from these functions is by querying their oracles. In other words, we assume that the attacker cannot break the encryption or hash primitives in any way; i.e., they are ideal cryptographic functions. We compute c according to the definition of convergent encryption: $c = E_{H(P)}(P)$. We allow an attacker to execute a program Σ containing a sequence of queries to the random oracles H, E, and E^{-1}, interspersed with arbitrary amounts of computation between each pair of adjacent queries. We refer to the count of queries in the program as the *length* of the program.

Theorem: Given ciphertext c, there exists no program Σ of length $O(n^\varepsilon)$ that can output plaintext P with probability $\Omega(1/n^\varepsilon)$ for any fixed ε, unless the attacker can *a priori* output P with probability $\Omega(1/n^{2\varepsilon})$.

Proof: Assume such a program exists. Let Σ' be the shortest such program. Σ' either does or does not include a query to oracle H for the value of $H(P)$. If it does not, then the $O(n^\varepsilon)$ queries to oracle H in program Σ' can, by eliminating alternatives to $H(P)$, increase the probability mass associated with $H(P)$ by a factor of at most $O(n^\varepsilon)$, so the probability for $H(P)$ is $o(1/n^\varepsilon)$. Thus the probability for $P = E^{-1}_{H(P)}(c)$ is also $o(1/n^\varepsilon) < \Omega(1/n^\varepsilon)$. Alternatively, if Σ' does include a query for $H(P)$, we can construct a new program Σ'' whose queries are a prefix of Σ' but which halts after a random query $q < |\Sigma'|$ and outputs the value of P queried on step $q + 1$ of program Σ'. With probability $\Omega(1/n^\varepsilon)$, Σ'' will output P, contradicting the assumption that Σ' is the shortest such program.

4. Identifying duplicate files – SALAD

Convergent encryption enables identical encrypted files to be recognized as identical, but there remains the problem of performing this identification across a large number of machines in a robust and decentralized manner. We solve this problem by storing file location and content information in a distributed data structure called a *SALAD*: a Self-Arranging, Lossy, Associative Database. For scalability, the file information is partitioned and dispersed among all machines in the system; and for fault-tolerance, each item of information is stored redundantly on multiple machines. Rather than using central coordination to orchestrate this partitioning, dispersal, and redundancy, SALAD employs simple statistical techniques, which have the unintended effect of making the database lossy. In our application, a small degree of lossiness is acceptable, so we have chosen to retain the (relative) simplicity of the system rather than to include additional machinery to rectify this lossiness.

4.1. SALAD record storage overview

Logically, a SALAD appears to be a centralized database. Each record in the database contains information about the location and content of one file. To add a new record to the database, a machine first computes a fingerprint of a file by hashing the file's (convergently encrypted) content and prepending the file size to the hash value. It then constructs a ⟨key, value⟩ record in which the key is the file's fingerprint and the value is the identifier of the machine where the file resides, and it inserts this record into the database. The database is indexed by fingerprint keys, so it can be associatively searched for records with matching fingerprints, thereby identifying and locating files with (probably) identical contents. (With 20-byte hash values, the probability that a set of F files contains even one pair of same-sized non-identical files with the same hash value is $F / 2^{20 \times 8 / 2} \approx F \times 10^{-24}$.)

Physically, the database is partitioned among all machines in the system. Within the context of SALAD, each machine is called a *leaf* (akin to a leaf in a tree data structure). Each record is stored in a set of local databases on zero or more leaves.

Leaves are grouped into *cells*, and all leaves within any given cell are responsible for storing the same set of records. Records are sorted into *buckets* according to the value of the fingerprint in each record. Each bucket of records is assigned to a particular cell, and all records in that bucket are stored redundantly on all leaves within the cell, as illustrated in Fig. 1. The number of cells grows linearly with the system size, and since the number of files also grows linearly with the system size, the expected number of records stored by each leaf is constant.

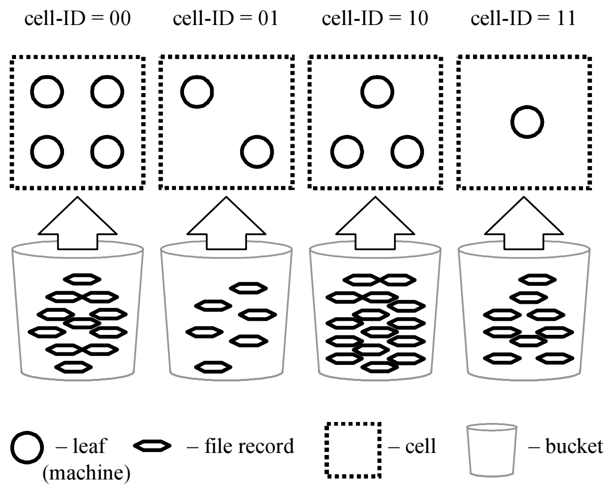


Fig. 1: Buckets of records in cells of leaves

A SALAD has two configuration parameters: its *target redundancy factor* Λ and its *dimensionality D*. Since each record is stored redundantly on all leaves in a particular cell, the degree of storage redundancy is equal to the mean number of leaves per cell. This value is known as the *actual redundancy factor* λ, and it is bounded (via the process described in Subsection 4.2) by the inequality:

$$\Lambda \le \lambda < 2\Lambda \tag{5}$$

For large systems, it is inefficient for each leaf to maintain a direct connection to every other leaf, so the leaves are organized (via the process described in Subsection 4.3) into a *D*-diameter directed graph. Each record is passed from leaf to leaf along the edges of the graph until, after at most *D* hops, it reaches the appropriate storage leaves.

4.2. SALAD partitioning and redundancy

The target redundancy factor Λ is combined with the *leaf count L* (the count of leaves in the SALAD, also called the *system size*) to determine a *cell-ID width W*, as follows (where the notation "lg" means binary logarithm):

$$W = \left\lfloor \lg \frac{L}{\Lambda} \right\rfloor \tag{6}$$

As described in Section 2 above, each leaf has a large (20-byte), unique identifier *i*. The least-significant *W* bits of a leaf's identifier or a record's fingerprint form a value called the *cell-ID* of that leaf or record. (For convenience, we sometimes use the term "identifier" to mean either a leaf's identifier or a record's fingerprint.) Formally, the cell-ID of identifier *i* is given by:

$$c(i) = i \bmod 2^W \tag{7}$$

Two identifiers are *cell-aligned* if their cell-ID values are equal. Cell-aligned leaves share the same cell, and records are stored on leaves with which they are cell-aligned, as illustrated in Fig. 1.

Before introducing the dimensionality parameter *D*, we describe the simplest SALAD configuration, in which *D* = 1, as in the example of Fig. 1. Each leaf in the SALAD maintains an estimate of the system size *L*. From this, it calculates *W* according to Eq. 6, and it computes cell-ID values for each leaf in the system (including itself) according to Eq. 7. Then, for each of its files, it hashes the file's content, creates a fingerprint record, and computes a cell-ID for the record. The leaf then sends each record to all leaves that it believes to be cell-aligned with the record. When each leaf receives a record, it stores the record if it considers itself to be cell-aligned with the record.

This example illustrates the statistical partitioning, redundancy, and lossiness of record storage. With no central coordination, records are distributed among all leaves, and records with matching fingerprints end up on the same set of leaves, so their identicality can be detected with a purely local search. Since machine identifiers and file content fingerprints are cryptographic hash values, they are evenly distributed, so the number of leaves on which each record is stored is governed by a Poisson distribution [21] with a mean of λ. Therefore, with probability $e^{-\lambda}$, a record will not be stored on any leaf.

Note that if two leaves have different estimates of the system size *L*, they may disagree about whether they are cell-aligned. However, this disagreement does not cause the SALAD to malfunction, only to be less efficient. If a leaf underestimates the system size, it may calculate an undersized cell-ID width *W*. With fewer bits in each cell-ID, cell-IDs are more likely to match each other, so the leaf may store more records than it needs to, and it may send records to leaves that don't need to receive them. If a leaf overestimates the system size, it may calculate an oversized cell-ID width *W*, which causes cell-IDs to be less likely to match each other, so the leaf may store fewer records than it needs to, and it may not send records to leaves that should receive them. Thus, an underestimate of *L* increases a leaf's workload, and an overestimate of *L* increases a leaf's lossiness.

Given *F* files in the system, the mean count of records stored by each leaf is *R*, calculated as follows:

$$R = \lambda \frac{F}{L} \tag{8}$$

Since $F \propto L$ and $\lambda < 2\,\Lambda$, *R* remains constant as the system size grows.

4.3. SALAD multi-hop information dispersal

Cells in a SALAD are organized into a *D*-dimensional hypercube. (Technically, it is a rectilinear hypersolid, since its dimensions are not always equal, but this term is cumbersome.) Coordinates in *D*-dimensional hyperspace are given with respect to *D* Cartesian axes. In two- or three-dimensional spaces, it is common to refer to these axes as the x-axis, y-axis, and z-axis, but for arbitrary dimensions, it is more convenient to use numbers instead of letters, so we refer to the 0-axis, the 1-axis, and so forth.

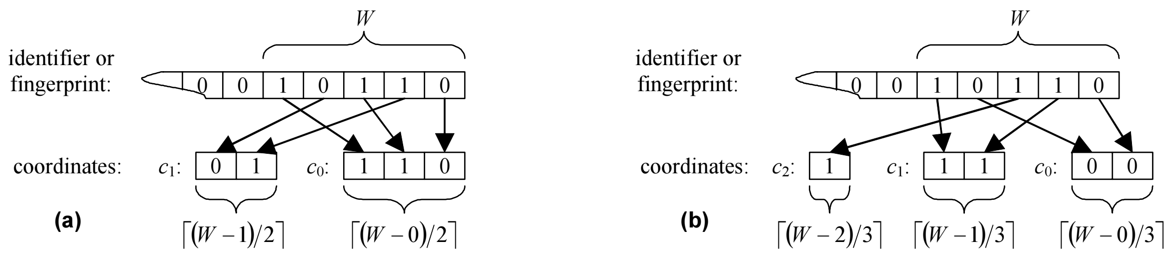


Fig. 2: Example extraction of cell-ID and coordinates from an identifier when (a) *D* = 2, (b) *D* = 3

Each cell-ID is decomposed into D *coordinates*, as illustrated in Fig. 2. Successive bits of each coordinate are taken from non-adjacent bits in the cell-ID so that when the system size L grows and the width of each coordinate consequently increases, the value of each coordinate undergoes minimal change. A cell's location within the hypercube is determined by the coordinates of its cell-ID. For example, in Fig. 2a, a leaf with the shown identifier has cell coordinates $c_0 = 6$ (110_2) and $c_1 = 1$ (01_2).

Formally, for $0 \le d < D$, the bit width W_d of the d-axis coordinate of an identifier is given by:

$$W_d = \left\lceil \frac{W-d}{D} \right\rceil \tag{9}$$

The d-axis coordinate of identifier i is defined by the following formula (where the notation $b_n(i)$ indicates the value of bit n in identifier i, and bit 0 is the LSB), which merely formalizes the procedure illustrated in Fig. 2:

$$c_d(i) = \sum_{k=0}^{W_d-1} 2^k \, b_{D\cdot k+d}(i) \tag{10}$$

Fig. 3 shows an example two-dimensional SALAD from the perspective of the black leaf. (Communication paths not relevant to the black leaf are omitted from this figure.) We refer to a row of cells that is parallel to any one of the Cartesian axes as a *vector* of cells. Two identifiers are *d-vector-aligned* if they are both in a vector of cells that runs parallel to the d-axis. This means that at least $D - 1$ of their coordinates match, but their d-axis coordinates might not. Formally:

$$a_d(i,j) \equiv \forall k \left[k \ne d \rightarrow c_k(i) = c_k(j) \right] \tag{11}$$

Identifiers are *vector-aligned* if they share any vector of cells. Thus, they are d-vector-aligned for some d, like leaves A and C in Fig. 3, but unlike A and E. Formally:

$$a(i,j) \equiv \exists d \left[a_d(i,j) \right] \tag{12}$$

Each leaf maintains a *leaf table* of all leaves that are vector-aligned with it, and these are the only leaves with which it communicates. The expected count of leaves in each vector is $\lambda\,(L/\lambda)^{1/D}$, so the mean leaf table size is T:

$$T = D\,\lambda \left(L/\lambda\right)^{1/D} - D\,\lambda + \lambda \approx D\,\lambda^{1-1/D}\,L^{1/D} \tag{13}$$

This is not very large. With L = 10,000, $\lambda = 3$, and $D = 2$, the mean leaf table size is about 350 entries.

After a new file record is generated, it makes its way through the salad by moving in one Cartesian direction per step, until after a maximum of D hops, it reaches leaves with which it is cell-aligned. A leaf performs the same set of steps either to insert a new record of its own into the SALAD or to deal with a record it receives from another leaf: In outline, each leaf determines the highest dimension d in which all of the (less-than-d)-axis coordinates of its own identifier equal those of the record's fingerprint. If $d < D$, then it forwards the fingerprint record along its d-axis vector to those leaves whose d-axis coordinates equal that of the record's fingerprint. After a maximum of D such hops, the fingerprint record will reach leaves that are cell-aligned with the fingerprint. When a leaf receives a cell-aligned record, the leaf stores the record in its local database, searches for records whose fingerprints match the new record's fingerprint, and notifies the appropriate machines if any matches are found.

For example, when $D = 2$, cells are organized into a square (a two-dimensional hypercube), and each leaf has entries in its leaf table for other leaves that are either in its horizontal vector or in its vertical vector. In Fig. 3, the black leaf has (binary) cell-ID $wxyz$ = 0110, and its coordinates are $c_0 = xz = 10$ and $c_1 = wy = 01$. Thus, it knows other leaves with cell-IDs $w1y0$ or $0x1z$, for any values of w, x, y, and z. (The figure shows directed connections to these known leaves via black arrows.)

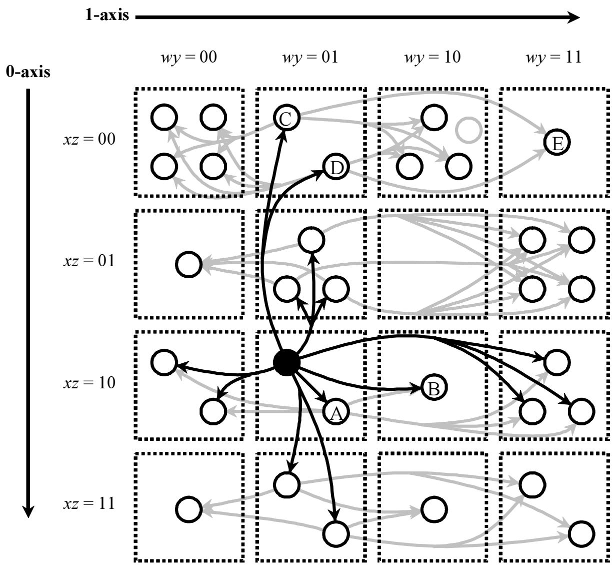


Fig. 3: SALAD from black leaf's perspective (*D*=2)

```
receive record ⟨$f$,$l$⟩                              // current leaf's identifier is $I$
  $d$ = 0
  while $d$ < $D$         // find highest dimension $d$ for which all (<d)-coordinates match
    if $c_d(f) \neq c_d(I)$
      send record ⟨$f$,$l$⟩ to all leaves $j$ for which $a_d(I,j) \wedge c_d(f) = c_d(j)$
      exit procedure                           // forward message along $d$ axis and exit
    $d$ = $d$ + 1
  // this leaf is cell-aligned with record's fingerprint
  if $l$ = $I$                                 // special case: this leaf generated record
    send record ⟨$f$,$l$⟩ to all leaves $j$ for which $c(f) = c(j)$
  search local database for records matching fingerprint f
  for each matching record ⟨$f$,$k$⟩
    notify machine $l$ that machine $k$ has file with fingerprint $f$
    notify machine $k$ that machine $l$ has file with fingerprint $f$
  store record ⟨$f$,$l$⟩ in local database
```

Fig. 4: Procedure for leaf *I* to insert record (*f,l*), locally initiated or upon receipt from another leaf

When the black leaf generates a record for one if its files, there are three cases: 1) If the record's cell-ID equals 0110, the leaf stores the record in its own database and sends it to the one other leaf in its cell, leaf A. 2) If the fingerprint cell-ID is $w1y0$ for $wy \neq 01$, then the 0-axis coordinates are equal but the 1-axis coordinates are not, so the black leaf sends the record along its 1-axis (horizontal) vector to leaves whose cell-ID equals $w1y0$. For example, if the fingerprint cell-ID is 1100, it is sent directly to leaf B. 3) If the fingerprint cell-ID is $wxyz$ for $xz \neq 10$, then the 0-axis coordinates are not equal, so the black leaf sends the record along its 0-axis (vertical) vector to leaves whose cell-ID equals $0x1z$. In this third case, if the fingerprint's 1-axis coordinate wy does not equal 01, then the recipient leaves will forward the record (horizontally, via the gray paths in the figure) to the appropriate leaves. For example, if the fingerprint cell-ID is 1010, it is sent to leaves C and D, who each forward it to leaf E.

Formally, the pseudo-code of Fig. 4 illustrates the procedure by which a leaf with identifier I inserts a new record $\langle f, l\rangle$ – indicating that leaf l contains a file with fingerprint f – into the SALAD.

Adding hops to the propagation of fingerprint records increases the system's lossiness. For a two-dimensional SALAD, a record will not be stored if it is not sent to any leaf on either the first or the second hop. When the system size L is very large, nearly all records require two hops, so the loss probability approaches $1 - (1 - e^{-\lambda})^2 \approx 2\, e^{-\lambda}$. In general, the loss probability for a D-dimensional salad is:

$$P_{\text{loss}} = 1 - \left(1 - e^{-\lambda}\right)^D \approx D\, e^{-\lambda} \qquad (14)$$

For example, with $\lambda = 3$ and $D = 2$, $P_{\text{loss}} \approx 10\%$.

4.4. SALAD maintenance – adding leaves

There are three aspects to maintaining a SALAD: Adding new leaves, removing dead leaves, and maintaining each leaf's estimate of the leaf count L. This subsection describes the first of these operations.

Each leaf is supposed to know all other leaves with which it is vector-aligned. Thus, when a machine is added to a SALAD as a new leaf, it needs to learn of all leaves that are vector-aligned with its identifier so it can add them to its leaf table, and these leaves need to add the new leaf to their leaf tables. The machine first discovers one or more arbitrary leaves in the SALAD by some out-of-band means (e.g. piggybacking on DHCP [1]). If the machine cannot find any extant leaves, it starts a new SALAD with itself as a singleton leaf. If the machine does find one or more leaves in a SALAD, it sends each of them a *join* message, and each of these messages is forwarded along D independent pathways of the hypercube until it reaches leaves that are vector-aligned with the new leaf's identifier. These vector-aligned leaves send *welcome* messages to the new leaf, which replies with *welcome-acknowledge* messages. These two types of messages cause the recipient to add an entry to its leaf table and to update its estimate of the system size L.

To formalize the join-message forwarding process, we generalize the concepts of cell-alignment and vector-alignment introduced in Subsections 4.2 and 4.3, respectively. Two identifiers are *δ-dimensionally-aligned* if they are both in the same δ-dimensional hypersquare of a SALAD. (For example, if there is any square that includes them both, they are two-dimensionally aligned, and if there is any vector that includes them both, they are one-dimensionally aligned.) This means that at least $D - \delta$ of their coordinates match. Formally:

$$a^{(\delta)}(i,j) \equiv \exists\Delta\left[|\Delta| = \delta \wedge \forall k\left[k \notin \Delta \rightarrow c_k(i) = c_k(j)\right]\right] \qquad (15)$$

Note that $a(i,j) \equiv a^{(1)}(i,j)$, so the term "vector-aligned" is a synonym for "one-dimensionally aligned"; similarly, the term "cell-aligned" is a synonym for "zero-dimensionally aligned". In Fig. 3, leaves C and D are 0-dimensionally aligned, leaves C and E are 1-dimensionally aligned, and leaves B and E are 2-dimensionally aligned.

We need to define one more term, namely the *effective dimensionality Đ* of a SALAD. A very small SALAD will not have enough leaves for even a minimal D-dimensional hypercube to contain a mean of Λ leaves per cell. For example, when $W = 1$, there are only two cells, irrespective of the value of D, so the salad is effectively one-dimensional. In general, when $W < D$, $W_d = 0$ for $W \leq d < D$. Thus, a SALAD's effective dimensionality is:

$$Đ = \min(W, D) \tag{16}$$

The cell for the new leaf is located at the intersection of *Đ* orthogonal vectors; the goal is for all leaves in these vectors to receive the new leaf's join message. The general strategy is for one leaf to send out *Đ* batches of messages, each of which is routed (by a series of hops through cells with progressively lower dimensional alignment) to one of the vectors that will contain the new leaf. When a leaf in one of the final vectors receives the join, it forwards it to all leaves in the vector. In Fig. 3, if a new leaf (shown as a gray circle) joins with cell-ID = 1000, and if it sends a join message to the black leaf, the black leaf will send a batch of one message to leaf B and a batch of two messages to leaves C and D. Leaf B will forward the join to all leaves in its column, and leaves C and D will each forward the join to all leaves in their row.

For this to work in a stateless fashion, the leaf that initiates the process must not be δ-dimensionally aligned with the new leaf for any $\delta <$ *Đ*. In Fig. 3, if leaf C initially received the new leaf's join message, it forwards the message to the leaves in cell 1000; however, these leaves will not forward the message to the other leaves in the column, because cell-aligned leaves are expecting to be at the end of the chain of forwarded join messages. Therefore, if the initially contacted leaf is δ-dimensionally aligned with the new leaf for some $\delta <$ *Đ*, it forwards a batch of join messages to a cell of leaves that are (δ+1)-dimensionally aligned, and this forwarding continues until the join message reaches leaves whose minimal dimensional alignment is *Đ*, which initiates the process described in the previous paragraph.

Formally, when leaf *I* receives a join message for a new leaf *n* from sender *s*, it performs the procedure shown in the pseudo-code of Fig. 5. In outline, the leaf determines the lowest dimensionality δ for which it is δ-dimensionally-aligned with the new leaf, and it determines the lowest dimensionality δ' for which the sender is δ'-dimensionally-aligned with the new leaf. (We recalculate this value at each leaf rather than passing it along with the message, because different leaves may have different

```
receive join ⟨s,n⟩                                              // my identifier is I
  δ = 0                             // δ is my lowest dimensional alignment with new leaf
  δ′ = 0                     // δ′ is sender's lowest dimensional alignment with new leaf
  Δ = ∅     // Δ is set of all dimensions in which I and n have non-matching coordinates
  for all d ∍ 0 ≤ d < Đ                                  // loop to calculate above values
    if $c_d(n) \neq c_d(I)$
      δ = δ + 1
      Δ = Δ ∪ {d}
    if ($c_d(n) \neq c_d(s)$)
      δ′ = δ′ + 1
  if (s = n)                                 // if join received directly from new leaf...
    δ′ = -1     // sender's dimensional alignment is considered lower than all others'
  if δ′ > δ                                     // sender has higher dimensional alignment
    if δ > 1       // if I am not vector-aligned, forward down one degree of alignment
      for all d ∈ {d′ ∍ d′ ∈ Δ ∧ (d′ + 1) mod Đ ∉ Δ}
        send join ⟨I,n⟩ to all known leaves j for which $c_d(n) = c_d(j)$
    else if δ = 1              // if I am vector-aligned, forward to leaves in my vector
      for all d ∈ Δ                        // |Δ| = δ = 1, so this loop only executes once
        send join ⟨I,n⟩ to all known leaves j for which $a_d(I,j)$
  else if δ′ < δ                                  // sender has lower dimensional alignment
    if δ < Đ    // if my dimensional alignment < Đ, forward up one degree of alignment
      for one $d \in_r$ {d′ ∍ 0 ≤ d′ < Đ ∧ d′ ∉ Δ}
        for one $c \in_r$ {c′ ∍ 0 ≤ c′ < $2^{W_d}$ ∧ c′ ≠ $c_d(n)$}
          send join ⟨I,n⟩ to all known leaves j for which $c_d(j) = c$
    else if δ > 1          // unless I'm vector-aligned, initiate Đ batches of messages
      for all d ∈ Δ
        send join ⟨I,n⟩ to all known leaves j for which $c_d(n) = c_d(j)$
    else                                            // I'm vector-aligned, and Đ = 1...
      send join ⟨I,n⟩ to all known leaves         // so forward join to everyone I know
  if δ < 2                                         // I'm vector aligned with new leaf...
    send welcome message to new leaf n                  // so welcome new leaf to SALAD
```

Fig. 5: Procedure for leaf *I* when receiving join message from sending leaf *s* to insert new leaf *n*

estimates of L.) The leaf then takes steps as described above to forward the join message to leaves of greater, lesser, or identical dimensional alignment, as appropriate. There are several annoying corner cases, which are handled appositely by the pseudo-code in Fig. 5.

As $L \to \infty$, each join message sent by the new leaf results in $D\,\lambda$ messages forwarded from the initially contacted D-dimensionally-aligned leaf to the (D–1)-dimensionally-aligned leaves. Each of these messages spawns λ additional messages for each dimensionality from D–2 down to 1. Each d-vector-aligned leaf at the end of a forwarding chain then forwards the message to $\lambda^{1-1/D} L^{1/D}$ d-vector-aligned leaves. Thus, each initially contacted leaf initiates a series of M messages for each new leaf, where M is:

$$M = D\,\lambda^{D-1/D}\,L^{1/D} \qquad (17)$$

When a vector-aligned (or cell-aligned) leaf receives a join message, it sends a welcome message to the new leaf. When the new leaf receives a welcome message from an extant leaf not in its leaf table, then if the extant leaf is vector-aligned with the new leaf, the new leaf takes three actions: 1) It adds the extant leaf to its leaf table; 2) it updates its estimate of the SALAD system size (see § 4.2.3); and 3) it replies to the extant leaf with a welcome-acknowledge message. When an extant leaf receives a welcome-acknowledge message from a new leaf, it performs the first two of these actions but does not reply with an acknowledgement.

4.5. SALAD maintenance – removing leaves

A new leaf must explicitly notify the SALAD that it wants to join, but a leaf can depart without notice, particularly if its departure is due to permanent machine failure. Thus, the SALAD must include a mechanism for removing stale leaf table entries. We employ the standard technique [e.g. 16] of sending periodic refresh messages between leaves, and each leaf flushes timed-out entries in its leaf table. In addition, a leaf that cleanly departs the SALAD sends explicit departure messages to all of the leaves in its leaf table.

4.6. SALAD maintenance – estimating leaf count

SALAD leaves use an estimate of the system size L to determine an appropriate value for the cell-ID width W. Since each leaf knows only the leaves for which it has entries in its leaf table, it has to estimate L based on the size T of its leaf table. The expected relationship between T and L is given by Eq. 13, so the leaf effectively inverts this equation. The actual procedure is a little more complicated, because a change to the estimated value of L can cause a change to the value of W, which in turn can cause leaves to be added to or removed from the leaf table, changing the value of T.

Whenever leaf I modifies its leaf table, it recalculates the appropriate cell-ID width W via the procedure illustrated in the pseudo-code of Fig. 6.

```
recalculate $W$                                          // $W$ is cell-ID width
  calculate known leaf ratio $r$                         // using Eq. 18
  $L = T / r$                                            // estimated leaf count
  $\Lambda' = \Lambda / (1 + \xi)$       // for decreasing width, use attenuated target redundancy
  $\hat{W} = \lfloor \lg (L / \Lambda') \rfloor$         // target cell-ID width
  while $\hat{W} < W$                    // while target width is less than actual width...
    $W = W - 1$                                          // decrement width
    request identifiers of newly vector-aligned leaves from neighbors
    recalculate $r$
    $L = T / r$
    $\hat{W} = \lfloor \lg (L / \Lambda') \rfloor$
  $\hat{W} = \lfloor \lg (L / \Lambda) \rfloor$      // for increasing width, use non-attenuated target redundancy
  while $\hat{W} > W$                    // while target width is greater than actual width
    $T'$ = count of remaining known leaves after increasing width
    $W' = W + 1$                                         // tentative cell-ID width
    $r'$ = visible leaf fraction based on $W'$           // tentative known leaf ratio
    $L' = T' / r'$                                       // tentative estimated leaf count
    $\hat{W}' = \lfloor \lg (L' / \Lambda) \rfloor$      // tentative target cell-ID width
    if $\hat{W}' < W'$               // if tentative target width less than tentative width...
      exit procedure    // tentative width is unstable, so exit without incrementing $W$
    $W = W'$                                             // increment $W$
    forget leaves that are no longer vector-aligned
    $r = r'$                        // update actual values to reflect tentative values
    $L = L'$
    $\hat{W} = \hat{W}'$
```

Fig. 6: Procedure followed by leaf *I* to recalculate cell-ID width *W* when leaf table size *T* changes

In outline, the leaf first calculates the ratio r of vector-aligned cells to total cells in the SALAD, based on the current value of W:

$$r = \frac{\sum_{d=0}^{D-1} 2^{W_d} - D + 1}{2^W} \tag{18}$$

Since the mean count of leaves per cell is a constant (λ), this ratio is also the expected ratio of leaves in its leaf table T to the total leaf count L. So from r and T, it estimates L, and it then calculates a target cell-ID width $\hat{W}$ according to Eq 1. If the target cell-ID width $\hat{W}$ is less than the current cell-ID width W, then the leaf decrements W, which folds the hypercube in half along dimension $d = (W - 1) \bmod D$. This increases the count of leaves with which it is d'-vector-aligned for all $d' \neq d$. To obtain identifiers and contact information for these leaves, the leaf requests this information from a set of leaves in its leaf table that should have these newly vector-aligned leaves in their leaf tables. In particular, it contacts λ leaves j with which it is d-vector aligned and for which $c_d(I) = c_d(j)$ when calculated according to the new (smaller) value of W but not according to the old (larger) value of W. The leaf then recalculates r, L, and $\hat{W}$ based on the new values of W and T, and it repeats the above process until the cell-ID width is less than or equal to the target cell-ID width.

If the target cell-ID width $\hat{W}$ is greater than the current cell-ID width W, the procedure is similar to the above in reverse; however, we have to be careful to prohibit an unstable state. Since increasing the cell-ID width causes leaves to be forgotten, it is possible that this will result in a smaller leaf table size T that yields a smaller system size estimate L that in turn produces a target cell-ID width $\hat{W}$ that is less than the newly increased cell-ID width W. Therefore, the leaf first calculates the leaf table size T' that would result if the cell-ID width were to be incremented. It then calculates tentative values W', r', L', and $\hat{W}'$ based on the tentative leaf table size T'. If the tentative target cell-ID width $\hat{W}'$ is less than the tentative new cell-ID width W', then the procedure exits instead of transiting to an unstable state. Otherwise, it increments W; forgets about leaves with which it is no longer vector-aligned; updates r, L, and $\hat{W}$ to reflect the new values of W and T; and repeats the above process until the cell-ID width is greater than or equal to the target cell-ID width.

The above procedure admits rapid fluctuation when the leaf table size vacillates around a threshold that separates two stable values of W. We prevent this fluctuation with hysteresis: When considering a decrease to the cell-ID width W, we calculate the target cell-ID width $\hat{W}$ using an attenuated target redundancy factor Λ':

$$\Lambda' = \frac{\Lambda}{1+\xi} \tag{19}$$

This is shown in the pseudo-code of Fig. 6. The damping factor ξ determines the spread between an L large enough to increase W and an L small enough to decrease W. The damping factor ξ should be relatively small, since it allows a reduction in the actual redundancy factor of the SALAD.

4.7. SALAD attack resilience

If SALAD were designed such that its leaves cooperatively determine the ranges of fingerprints that each leaf stores, it might be possible for a set of malicious leaves to launch a targeted attack against a particular range of values, by arranging for themselves to be the designated record stores for this range. However, because of the SALAD's purely statistical construction, such an attack is greatly limited: Each leaf determines its fingerprint range independently from the ranges of all other leaves, so the most damage a malicious leaf can do is to decrease the overall redundancy of the system.

For $D > 1$, it is possible to target an attack, but only in a fairly weak way. By choosing their own identifiers to be vector-aligned with a victim leaf, a set of m malicious leaves can increase the size of the victim's leaf table, thereby increasing its system size estimate L, which increases the leaf's lossiness as described at the end of Subsection 4.2. The effective redundancy factor λ' for the victim leaf's records will be:

$$\lambda' = \lambda \left(1 - \frac{m}{L}\right)^D \tag{20}$$

Thus, not only does increasing a SALAD's dimensionality increase the loss probability for a given redundancy factor (Eq. 14), but also it increases the susceptibility of the system to attack. We therefore suggest constructing a SALAD with a dimensionality no higher than that needed to achieve leaf tables of a manageably small size.

5. Simulation evaluation

Since the current implementation of Farsite is not complete or stable enough to run on a corporate network, we evaluated the DFC subsystem via large-scale simulation using file content data collected from 585 desktop file systems. We distributed a scanning program to a randomly selected set of Microsoft employees and asked them to scan their desktop machines. The program computed a 36-byte cryptographically strong hash of each 64-Kbyte block of all files on their systems, and it recorded these hashes along with file sizes and other attributes. The scanned systems contain 10,514,105 files in 730,871 directories, totaling 685 GB of file data. There were 4,060,748 distinct file contents totaling 368 GB of file data, implying that coalescing duplicates could ideally reclaim up to 46% of all consumed space.

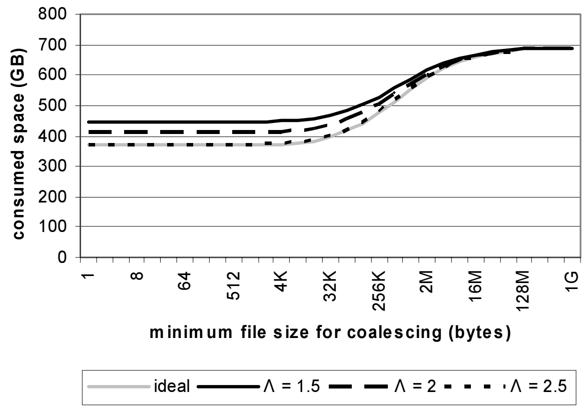


Fig. 7: Consumed space vs. minimum file size

We ran a two-dimensional DFC system on 585 simulated machines, each of which held content from one of the scanned desktop file systems. The SALAD was initialized with a single leaf, and the remaining 584 machines were each added to the SALAD by the procedure outlined in Subsection 4.4. We recorded the sizes of each machine's leaf table and fingerprint database, as well as the number of messages exchanged.

By setting a threshold on the minimum file size eligible for coalescing, we can substantially reduce the message traffic and fingerprint database sizes. Fig. 7 shows the consumed space in the system versus this minimum size. The effect on space consumption is negligible for thresholds below 4 Kbytes. This figure also shows that a target redundancy factor of Λ = 2.5 achieves nearly all possible space reclamation.

We tested the resilience of the DFC system to machine failure by randomly failing the simulated machines. Fig. 8 shows the consumed space versus machine failure probability. With Λ = 2.5, even when machines fail half of the time, the system can still reclaim 38% of used space, comparing favorably to the optimal value of 46%.

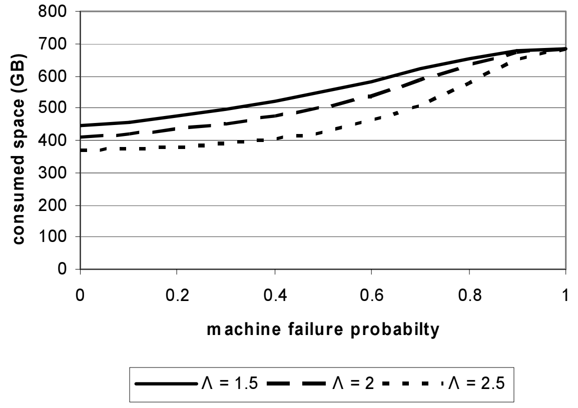


Fig. 8: Consumed space vs. machine failure rate

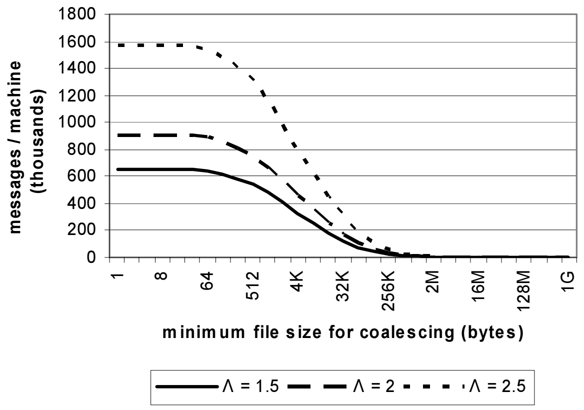


Fig. 9: Message count vs. minimum file size

Fig. 9 shows the mean count of messages each machine sent and received versus the minimum file-size threshold. By setting this threshold to 4 Kbytes, the mean message count is cut in half without measurably reducing the effectiveness of the system (cf. Fig. 7). Fig. 10 shows the cumulative distribution of machines by count of messages sent and received, with no minimum file-size threshold. The coefficients of variation [21] CoV_{Λ} for these curves are $\mathrm{CoV}_{1.5} = 0.64$, $\mathrm{CoV}_{2.0} = 0.39$, and $\mathrm{CoV}_{2.5} = 0.39$. Combined with the smooth shape of the curves, these values imply that the machines share the communication load relatively evenly, especially as Λ is increased.

Fig. 11 shows the mean database size on each machine versus the minimum file-size threshold. As with the message count, setting this threshold to 4 Kbytes halves the mean database size. Fig. 12 shows the cumulative distribution of machines by database size, with no minimum file-size threshold. The coefficients of variation are $\mathrm{CoV}_{1.5} = 0.28$, $\mathrm{CoV}_{2.0} = 0.31$, and $\mathrm{CoV}_{2.5} = 2.4\times10^{-5}$, which are fairly small. However, all three curves show bimodal distributions: For Λ = 1.5 and Λ = 2.0, most machines store fewer than 40,000 records but a substantial minority store nearly 80,000 records. For Λ = 2.5, the

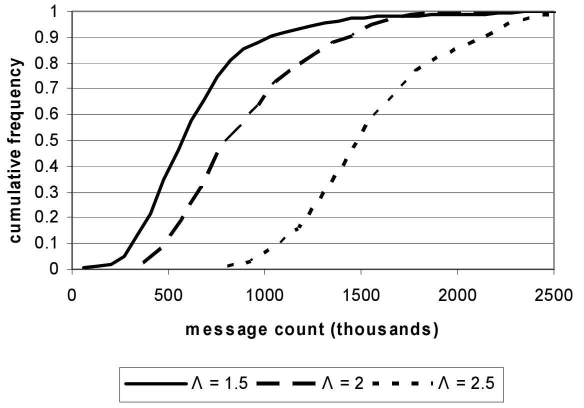


Fig. 10: CDF of machines by message count

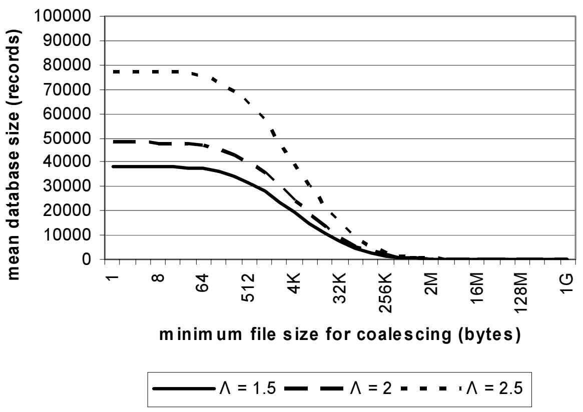


Fig. 11: Database size vs. minimum file size

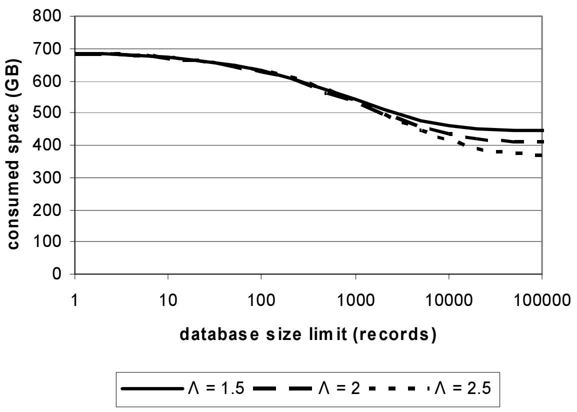


Fig. 13: Consumed space vs. database size limit

reverse is true. This suggests that the differences in storage load among machines is due primarily to slight variations in machines' estimates of L, filtered through the step discontinuity in the calculation of W (cf. Eq. 6), rather than due to non-uniformity in the distribution of machine identifiers or file content fingerprints.

The substantial skew in Fig. 12 reveals an opportunity for reducing the record storage load in the system. We ran a set of experiments in which we limited the database size on each machine. When a machine receives a record that it should store, if its database size limit has been reached, it discards a record in the database with the lowest fingerprint value (corresponding to the smallest file) and replaces it with the newly received record. If no record in the database has a lower fingerprint value than the new record, the machine discards the new record (after forwarding it if appropriate). Fig. 13 shows the consumed space versus the database size limit. A limit of 40,000 records makes no measurable difference in the consumed space for any Λ. For Λ = 2.5, even with a limit of 8000 records (an order of magnitude smaller than the mean database size), the system can still reclaim 38% of used space, compared to the optimum of 46%.

For our final experiment, we started with a singleton SALAD and incrementally increased the system size up to 10,000 simulated machines. Fig. 14 shows the mean leaf table size versus system size. The square-root relationship predicted by Eq. 13 is evident in these curves, as is a periodic variation due to the discretization of W.

Fig. 15 shows the cumulative distribution of machines by leaf table size for L = 585 (the system size for our earlier experiments) and for L = 10,000 (the system size for our last experiment). When Λ = 1.5, the lossiness is so high that a significant (if small) fraction of machines have nearly empty leaf tables. For larger values of Λ, the curves show that there is reasonably close agreement among machines about the estimated system size, except in the case where Λ = 2.5 and L = 10,000. In this case, lg(L/Λ) is very close to an integer, so slight variations in machines' estimates of L cause Eq. 6 to produce different cell-ID widths for different machines, yielding this bimodal distribution.

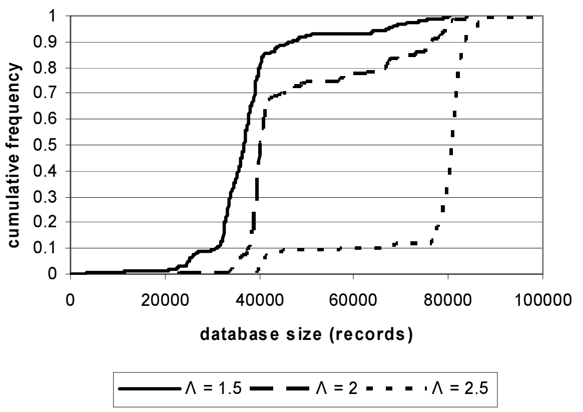


Fig. 12: CDF of machines by database size

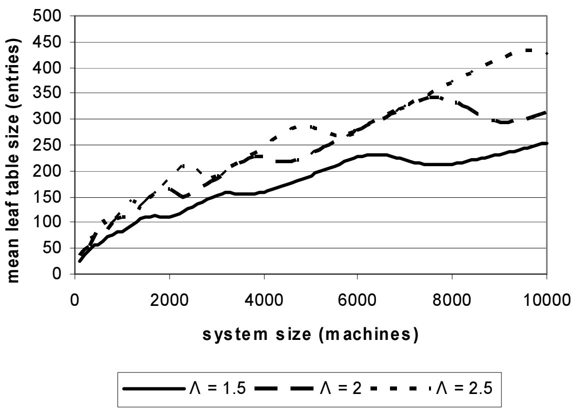
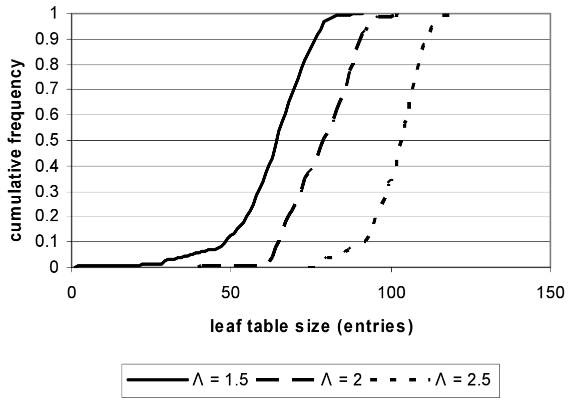
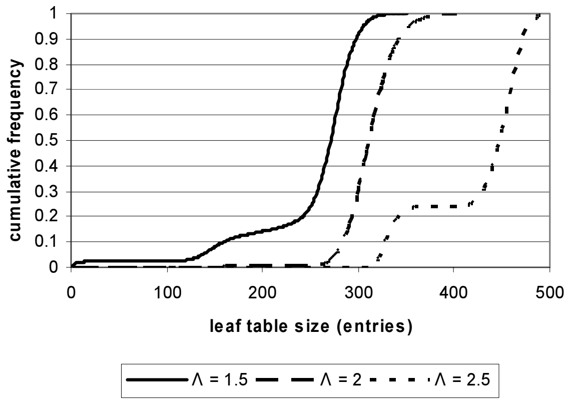


Fig. 14: Leaf table size vs. system size



(a)



(b)

Fig. 15: CDF of machines by leaf table size for (a) *L* = 585, (b) *L* = 10,000

6. Related work

To our knowledge, coalescing of identical files is not performed by any distributed storage system other than Farsite. The resulting increase in available space could benefit server-based distributed file systems such as AFS [20] and Ficus [19], serverless distributed file systems such as xFS [2] and Frangipani [37], content publishing systems such as Publius [38] and Freenet [12], and archival storage systems such as Intermemory [18].

Windows® 2000 [34] has a Single-Instance Store [7] that coalesces identical files within a local file system.

LBFS [28] identifies identical portions of different files to reduce network bandwidth rather than storage usage.

Convergent encryption deliberately leaks information. Other research has studied unintentional leaks through side channels [22] such as computational timing [23], measured power consumption [24], or response to injected faults [5]. Like convergent encryption, BEAR [3] derives an encryption key from a partial plaintext hash. Song et al. [35] developed techniques for searching encrypted data.

SALAD has similarities to the distributed indexing systems Chord [36], Pastry [31], and Tapestry [40], all of which are based on Plaxton trees [29]. These systems use $O(\log n)$-sized neighbor tables to route information to the appropriate node in $O(\log n)$ hops. Also similar is CAN [30], which uses $O(d)$-sized neighbor tables to route information to nodes in $O(d\, n^{1/d})$ hops. SALAD complements these approaches by using $O(d\, n^{1/d})$-sized neighbor tables to route in $O(d)$ hops. These other systems are not lossy, but they appear less immune to targeted attack than SALAD is. SALAD's configurable lossiness is similar to that of a Bloom filter [6], although it yields false negatives rather than false positives.

Farsite relocates identical files to the same machines so their contents may be coalesced. Other research on file relocation has been to balance the load of file access [9, 39] to migrate replicas near points of high usage [10, 17, 25, 33], or to improve file availability [14, 26].

7. Summary

Farsite is a distributed file system that provides security and reliability by storing encrypted replicas of each file on multiple desktop machines. To free space for storing these replicas, the system coalesces incidentally duplicated files, such as shared documents among workgroups or multiple users' copies of common application programs.

This involves a cryptosystem that enables identical files to be coalesced even if encrypted with different keys, a scalable distributed database to identify identical files, a file-relocation system that co-locates identical files on the same machines, and a single-instance store that coalesces identical files while retaining separate-file semantics.

Simulation using file content data from 585 desktop file systems shows that the duplicate-file coalescing system is scalable, highly effective, and fault-tolerant.

References

[1] S. Alexander and R. Droms, “DHCP Options and BOOTP Vendor Extensions”, RFC 2132, Mar 1997.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, “Serverless Network File Systems”, 15th SOSP, ACM, Dec 1995, pp. 109-126.

[3] R. Anderson and E. Biham, “Two Practical and Provably Secure Block Ciphers: BEAR and LION”, 3rd International Workshop on Fast Software Encryption, 1996, pp. 113-120.

[4] M. Bellare and P. Rogaway, “Random Oracles are Practical: A Paradigm for Designing Efficient Protocols”, 1st Conference on Computer and Communications Security, ACM, 1993, pp. 62-73.

[5] E. Biham and A. Shamir, “Differential Fault Analysis of Secret Key Cryptosystems”, CRYPTO ’91, 1991, pp. 156-171.

[6] B. H. Bloom, “Space/Time Trade-Offs in Hash Coding with Allowable Errors”, CACM 13(7), Jul 1970, pp. 422-426.

[7] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur, “Single Instance Storage in Windows® 2000”, 4th Windows Systems Symposium, USENIX, 2000, pp. 13-24.

[8] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, “Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs”, SIGMETRICS 2000, ACM, 2000, pp. 34-43.

[9] A. Brinkmann, K. Salzwedel, and C. Scheideler, “Efficient, Distributed Data Placement Strategies for Storage Area Networks”, 12th SPAA, ACM, Jun 2000.

[10] G. Cabri, A. Corradi, and F. Zambonelli, “Experience of Adaptive Replication in Distributed File Systems”, 22nd EUROMICRO, IEEE, Sep 1996, pp. 459-466.

[11] M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance”, 3rd OSDI, USENIX, Feb 1999, pp. 173-186.

[12] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, “Freenet: A Distributed Anonymous Information Storage and Retrieval System”, ICSI Workshop on Design Issues in Anonymity and Unobervability, Jul 2000.

[13] J. R. Douceur and W. J. Bolosky, “A Large-Scale Study of File-System Contents”, SIGMETRICS ’99, ACM, May 1999, pp. 59-70.

[14] J. R. Douceur and R. P. Wattenhofer, “Optimizing File Availability in a Secure Serverless Distributed File System”, 20th SRDS, IEEE, Oct 2001, pp. 4-13.

[15] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, “Reclaiming Space from Duplicate Files in a Serverless Distributed File System”, 22nd International Conference on Distributed Computing Systems, IEEE, July 2002.

[16] R. Droms, “Dynamic Host Configuration Protocol”, RFC 2131, Mar 1997.

[17] B. Gavish and O. R. Liu Sheng, “Dynamic File Migration in Distributed Computer Systems”, CACM 33 (2), ACM, Feb 1990, pp. 177-189.

[18] A. Goldberg and P. Yianilos, “Towards an Archival Intermemory”, International Forum on Research and Technology Advances in Digital Libraries, IEEE, Apr 1998, pp. 147-156.

[19] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier, “Implementation of the Ficus Replicated File System”, 1990 USENIX Conference, Usenix, Jun 1990, pp. 63-71.

[20] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and Performance in a Distributed File System,” Transactions on Computer Systems, ACM, 1988, pp. 51-81.

[21] R. Jain. The Art of Computer Systems Performance Analysis. John Wiley & Sons, 1991.

[22] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, “Side Channel Cryptanalysis of Product Ciphers”, Journal of Computer Security 8(2-3), 2000, pp. 141-158.

[23] P. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems”, CRYPTO '96, 1996, pp. 104-113.

[24] P. Kocher, J. Jaffe, and B. Jun. “Differential Power Analysis”, CRYPTO '99, 1999, pp. 388-397.

[25] Ø. Kure, “Optimization of File Migration in Distributed Systems”, Technical Report UCB/CSD 88/413, University of California at Berkeley, Apr 1988.

[26] D. L. McCue and M. C. Little, “Computing Replica Placement in Distributed Systems”, 2nd Workshop on Management of Replicated Data, IEEE, Nov 1992, pp. 58-61.

[27] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997.

[28] A. Muthitacharoen, B. Chen, and D. Mazières, “A Low-Bandwidth Network File System”, (to appear) 18th SOSP, ACM, Oct 2001.

[29] C. G. Plaxton, R. Rajaraman, and A. W. Richa, “Accessing Nearby Copies of Replicated Objects in a Distributed Environment”, 9th SPAA, ACM, Jun 1997, pp. 311-320.

[30] S. Ratnasamy, P. Francis, M. Handley, and R. Karp, “A Scalable Content-Addressable Network”, SIGCOMM 2001, ACM, Aug 2001.

[31] A. Rowstron and P. Druschel, “Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems”, (SIGCOMM 2001 submission).

[32] S. Saroiu, P. K. Gummadi, and S. D. Gribble, “A Measurement Study of Peer-to-Peer File Sharing Systems”, Multimedia Computing and Networking 2002, SPIE, 2002, (to appear).

[33] A. Siegel, K. Birman, and K. Marzullo, “Deceit: A Flexible Distributed File System”, Summer 1990 USENIX Conference, USENIX, Jun 1990.

[34] D. Solomon, Inside Windows NT, 2nd Ed., MS Press, 1998.

[35] D. X. Song, D. Wagner, and A. Perrig, “Practical Techniques for Searches on Encrypted Data”, IEEE Symposium on Security and Privacy, 2000, pp. 44-55.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, SIGCOMM 2001, ACM, Aug 2001.

[37] C. Thekkath, T. Mann, and E. Lee, “Frangipani: A Scalable Distributed File System”, 16th SOSP, ACM, Dec 1997, pp. 224-237.

[38] M. Waldman, A. D. Rubin, and L. F. Cranor, “Publius: A Robust, Tamper-Evident Censorship-Resistant Web Publishing System”, 9th USENIX Security Symposium, Aug 2000, pp. 59-72.

[39] J. Wolf, “The Placement Optimization Program: A Practical Solution to the Disk File Assignment Problem”, SIGMETRICS ’89, ACM, May 1989.

[40] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, “Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing”, UCB Tech Report UCB/CSD-01-1141.