

# Pastiche: Making Backup Cheap and Easy

Landon P. Cox, Christopher D. Murray, and Brian D. Noble  
Department of Electrical Engineering and Computer Science  
University of Michigan, Ann Arbor, MI 48109-2122  
{lpcox,cdmurray,bnoble}@umich.edu <http://mobility.eecs.umich.edu/>

## Abstract

Backup is cumbersome and expensive. Individual users almost never back up their data, and backup is a significant cost in large organizations. This paper presents *Pastiche*, a simple and inexpensive backup system. *Pastiche* exploits excess disk capacity to perform peer-to-peer backup with no administrative costs. Each node minimizes storage overhead by selecting peers that share a significant amount of data. It is easy for common installations to find suitable peers, and peers with high overlap can be identified with only hundreds of bytes. *Pastiche* provides mechanisms for confidentiality, integrity, and detection of failed or malicious peers. A *Pastiche* prototype suffers only 7.4% overhead for a modified Andrew Benchmark, and restore performance is comparable to cross-machine copy.

## 1 Introduction

Backup is cumbersome and expensive. Personal machines are backed up rarely, if at all. Internet backup services exist, but are costly. For example, Connected TLM offers individual users backup of up to 4 GB—but covers neither applications nor the operating system—for \$15 per month [15]. Machines within an organization can be backed up centrally, but at significant cost. For example, the computer support arm of Michigan’s College of Engineering provides backup service of up to 8 GB on a single machine for \$30 per month.

The cost and inconvenience of backup are unavoidable, and often prohibitive. Small-scale solutions require significant administrative efforts. Large-scale solutions require aggregation of substantial demand to justify the capital costs of a large, centralized repository.

There is increasing recognition that disks better serve the needs of near-line archival storage. The purchasing cost of disk subsystems has caught up with tape [33], and disks provide better access and restore time. At the same time, the conventional wisdom that data expands to fill storage space is proving to be untrue.

Douceur’s examination of nearly five thousand machines finds that file systems are now only 53% full, on average [19]. Furthermore, the amount of newly written data per client per day is a small fraction of the total file system [43, 45, 50]. Several systems take advantage of low write traffic in the presence of excess storage capacity, including Elephant [43] and S4 [47].

*Pastiche* uses some of this excess disk capacity for efficient, effective, and administration-free backup. *Pastiche* nodes form a cooperative—though untrusted—collection of machines that provide mutual backup services. Because individual machines may come and go [6], each *Pastiche* node must replicate its archival data on more than one peer. Most of these replicas are placed nearby to ease network overhead and minimize restore time, though at least one replica must be elsewhere to guard against catastrophe. With no effort on the part of the user and modest additional disk space, backups are provided automatically. *Pastiche* is primarily aimed at end-user machines, but it can be used for back-end repositories with some care.

*Pastiche* cannot afford to keep duplicate copies of data on each replica. Luckily, much of the data on a given machine is not unique, and is generated at install time. Furthermore, for most machines, common data will be shared widely. The default installation of Office 2000 Professional requires 217 MB; it is nearly ubiquitous and different installations are largely the same. Randomly grouping disparate file systems and coalescing duplicate files produces significant savings [5]. *Pastiche* identifies systems with overlap to increase this savings.

*Pastiche* builds on three recent developments to accomplish its goals. Pastry [41], a peer-to-peer network, provides scalable, self-administered routing and node location. Content-based indexing [27, 30] provides flexible discovery of redundant data within similar files. Convergent encryption [6] allows hosts to use the same encrypted representation for common data without sharing keys.

Even with these building blocks, *Pastiche* still faces a number of challenges. How can nodes discover *backup buddies* with substantial overlap without a centralized

directory? How can nodes reuse their own on-disk state to backup others? How can nodes restore files—or an entire machine—without requiring administrative intervention? How can nodes detect unfaithful buddies?

Pastiche computes a small *abstract* of a file system’s content that potential backup buddies can inspect to approximate overlap. Pastiche is able to limit the size of the abstract by taking advantage of the fact that arbitrary, small pieces of larger logical entities are almost always unique and can, therefore, stand for the whole. This allows machines with common installations to find suitable buddies with very little effort. Machines with uncommon installations may need to use a Pastry overlay with a new routing metric, *coverage rate*.

Because sharing is supported at a sub-file granularity, Pastiche provides a new file system, *chunkstore*. Chunkstore stores all data—the host’s as well backup state—in the units of sharing, without compromising the performance of common-case workloads.

Archive state is described by a *skeleton* tree of meta-data. The root of this tree can be recovered from the Pastry overlay with only the name and passphrase of the machine to be restored. Entire file systems are restored as easily as a single file.

To address the problem of storing data on untrusted nodes, Pastiche uses a probabilistic mechanism to detect missing backup state by periodically querying buddies for stored data. Pastiche is able to keep the overhead of these queries small, bounding the chance of loss.

An examination of file system data shows that abstracts of a few hundred bytes effectively discriminate between candidate buddies. Simulations show that Pastiche nodes with common installations can easily find others with good overlap. The chunkstore file system induces overhead of 7.4% on a modified Andrew Benchmark, despite its unoptimized layout. Finally, analytical results show that a Pastiche node can detect corrupted backup state with high probability by checking about 0.1% of all chunks.

## 2 Enabling Technologies

Pastiche depends on three enabling technologies. The first is Pastry, a scalable, self-organizing, peer-to-peer routing and object location infrastructure [41]. The second is content-based indexing [27, 30], a technique that finds common data across different files. The third is convergent encryption [6], which allows sharing without compromising privacy. The remainder of this section describes each of these, focusing on the features essential to Pastiche.

### 2.1 Peer-to-Peer Routing

Pastiche eschews the use of a centralized authority to manage backup sites. Such an authority would be a single point of control, limiting scalability and increasing expense. Instead, Pastiche relies on Pastry, a scalable, self-organizing, routing and object location infrastructure for peer-to-peer applications.

Each Pastry node is named by a *nodeId*; the set of all *nodeId*’s are expected to be uniformly distributed in the *nodeId* space. Any two Pastry nodes have some way of measuring their *proximity* to one another. Typically, this metric captures some notion of network costs.

Each node  $N$  maintains three sets of state: a *leaf set*, a *neighborhood set*, and a *routing table*. The leaf set consists of  $L$  nodes;  $L/2$  are those with the closest numerically smaller *nodeIds*, and  $L/2$  are the closest larger ones. The neighborhood set of  $M$  nodes contains those closest to  $N$  according to the proximity metric. The Pastry group has deprecated the neighborhood set. However, as we show in Section 5.3, the neighborhood set is critical to buddy discovery for nodes with uncommon installations.

The routing table supports *prefix routing*. There is one row per hexadecimal digit in the *nodeId* space. The first row contains a list of nodes whose *nodeIds* differ from the current node’s in the first digit; there is one entry for each possible digit value. The second row holds a list of nodes whose first digit is the same as the current node’s, but whose second digit differs. To route to an arbitrary destination, a packet is forwarded to the node with a matching prefix that is at least one digit longer than that of the current node. If such a node is not known, the packet is forwarded to a node with an identical prefix, but that is numerically closer to the destination in *nodeId* space. This process continues until the destination node appears in the leaf set, after which it is delivered directly. The expected number of routing steps is  $\log N$ , where  $N$  is the number of nodes.

Many positions in the routing table can be satisfied by more than one node. When given a choice, Pastry records the closest node according to the proximity metric. As a result, the nodes in a routing table sharing a shorter prefix will tend to be nearby since there are many such nodes. However, any particular node is likely to be far.

Pastry is self-organizing; nodes can come and go at will. To maintain Pastry’s locality properties, a new node must join with one that is nearby according to the proximity metric. Pastry provides a *seed discovery protocol* that finds such a node given an arbitrary starting point [10]. Pastiche uses two separate Pastry overlay networks, but uses them only during buddy discovery.

Once a node has identified its backup set, all further traffic is routed directly via IP.

Pastiche adds two mechanisms to Pastry. The first is a technique called the *lighthouse sweep* that guarantees that distinct Pastry nodes are queried during buddy discovery. The second is a distance metric based on file system contents; this is used to find buddies for machines with rare installations.

## 2.2 Content-Based Indexing

To minimize storage overhead, Pastiche must find redundant data across versions of files, files in a system, and files on distinct machines. Rsync [48] and Tivoli [1] employ schemes for finding common subsets in two versions of (presumably) the same file. However, these techniques cannot easily capture general sharing.

The challenge is to find sharing—and hence structure—across seemingly unrelated files *without* knowing the underlying structure. Content-based indexing accomplishes this by identifying boundary regions, called *anchors* [27], using Rabin fingerprints [39]. A fingerprint is computed for each overlapping  $k$ -byte substring in a file. If the low-order bits of a fingerprint match a predetermined value, that offset is marked as an anchor. Anchors divide files into *chunks*. Since anchors are purely content-driven, editing operations change only the chunks they touch, even if they change offsets.

As with LBFS [30], Pastiche names each chunk by taking a SHA-1 hash [32] of its contents. The probability that two different chunks will hash to the same value is much lower than the probability of hardware errors. It is therefore customary to assume that chunks that hash to the same value are in fact the same chunk, and Pastiche adopts this custom. In Pastiche, these chunks form the basis of on-disk file structures in chunkstore to easily share data between local host and remote client.

## 2.3 Sharing with Confidentiality

A well-chosen backup buddy has much of a Pastiche node’s data, even before the first backup. However, Pastiche must guarantee the confidentiality and integrity of its participants’ data. If clients are free to choose their own cryptographic keys, chunks with identical content will be represented differently, precluding sharing.

The Farsite file system solves this problem with convergent encryption [6]. Under convergent encryption, each file is encrypted by a key derived from the file’s contents. Farsite then encrypts the file key with a *key-encrypting key*, known only to the client; this key is stored with the file. As a file is shared by more clients, it gains new encrypted keys; each client shares the single encrypted file.

Pastiche applies convergent encryption to all on-disk chunks. If a Pastiche node backs up a new chunk not already stored on a backup buddy, the buddy cannot discover its contents after shipment. However, if the buddy has that chunk, it knows that the node also stores that data. Pastiche allows this small information leak in exchange for increased storage efficiency.

## 3 Design

Pastiche data is stored on disk as chunks. Chunk boundaries are determined by content-based indexing, and encrypted with convergent encryption. Chunks carry *owner lists*, which name the set of nodes with an interest in a chunk. Chunks may be stored on a machine’s disk for that machine, a backup client, or both. Data chunks are immutable, and each chunk persists until no node holds a reference to it. Pastiche ensures that only rightful owners are capable of removing a reference to (and possibly deleting) a chunk.

When a newly written file is closed, it is scheduled for chunking. Each chunk  $c$  is hashed; the result is called the chunk’s *handle*,  $H_c$ . Each handle is used to generate a symmetric encryption key,  $K_c$ , for its chunk. The handle is hashed again to determine the public chunkId,  $I_c$ , of the chunk. Each chunk is stored on disk encrypted by  $K_c$  and named by  $I_c$ . This process is illustrated in Figure 1.

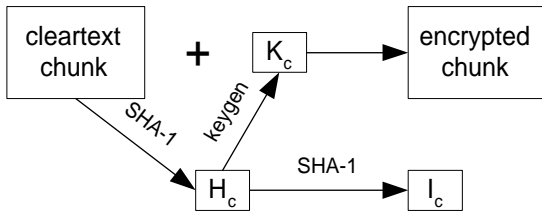
Before writing a chunk to disk, Pastiche first checks to see if it already exists. If so, the local host is added to the owner list if necessary, and the local reference count is incremented. Otherwise, the chunk is encrypted, a message authentication code [31] is appended, and the chunk is written out to disk with a reference count of one for the local owner.

Chunking and writing to disk are deferred to avoid needless overhead for files with short lifetimes [50], at the cost of slightly weaker persistence guarantees. The list of chunkIds that describes a node’s current file system is called its *signature*.

Data chunks are immutable. When a file is overwritten, its set of constituent chunks may change. Any chunks no longer part of the file have their local owner’s reference count decremented; if the reference count drops to zero, the local owner is removed. If the owner list becomes empty, the chunk’s storage is reclaimed. File deletion is handled similarly.

The meta-data for a file contains the list of handles for the chunks comprising that file, plus the usual contents: ownership, permissions, creation and modification times, etc. The handles in this list are used to derive the decryption key and chunkId for each constituent chunk.

Meta-data chunks are encrypted to protect the handle values and hence cryptographic keys. This differs slightly from Farsite’s use of convergent encryption.



This figure depicts how chunks are stored and named. A cleartext chunk is hashed, producing its handle. The handle is used for key generation, and hashed again to produce the chunkId. The chunk is stored encrypted by the key and named by the chunkId.

Figure 1: Naming and Storing Chunks

Farsite stores keys with data, encrypting each derived key with a key private to the writing host. Pastiche stores handles, and hence keys, in the meta-data blocks.

Unlike data, meta-data is not chunked and is mutable. Pastiche does not chunk meta-data because it is typically small and unlikely to be shared. Meta-data is mutable to avoid cascading writes. Each write to a file changes its constituent chunkIds. If meta-data were immutable, Pastiche would have to create a new meta-data chunk with a new name for every update. This new name would have to be added to the enclosing directory, which would also change, and so on to the file system root. Instead, the  $H_c$ ,  $K_c$ , and  $I_c$  for a file’s meta-data are computed only at creation time, and are re-used thereafter.

The meta-data object corresponding to a file system root is treated specially: its  $H_c$  is generated by a host-specific passphrase. As Section 3.4 explains, this passphrase plus the machine’s name is all that is required to restore a machine from scratch.

A chunk that is part of another node’s backup state includes that nodeId in its owner list. Remote hosts supply a public key with their backup storage requests. Requests to remove references must be signed by the corresponding secret key, otherwise those requests are rejected. This prevents third-party deletions, though it does not prevent the buddy from dropping chunks of its own accord.

Storing files directly as chunks simplifies a number of Pastiche’s tasks and imposes modest performance costs. It simplifies the implementation of chunk sharing, convergent encryption, and backup/restore. Without chunk-store, Pastiche would have to keep a persistent index consistent with on-disk files. This index would have to be consulted during backup and restore, and complicates garbage collection of chunks retired during snapshot. Furthermore, convergent encryption requires that each chunk be encrypted separately, complicating a contiguous layout. The only alternative would be to detect sharing only at the file level, with a corresponding increase in storage costs for backup.

### 3.1 Abstracts: Finding Redundancy

Much of the long-lived data on a machine is written once and then never overwritten. Observations of file type [19] and volume ownership [45] suggest that the amount of data written thereafter will be small. In other words, the signature of a node is not likely to change much over time. Therefore, if all data had to be shipped to a backup site, the initial backup of a freshly installed machine is likely to be the most expensive.

An ideal backup buddy for a newly-installed Pastiche node is one that holds a superset of the new machine’s data; machines with more complete coverage are preferred to those with less. One simple way to find such nodes is to ship the full signature of the new node to candidate buddies, and have them report degree of overlap.

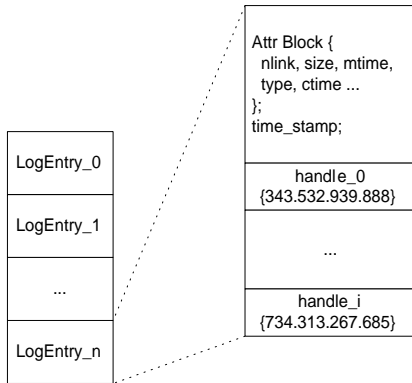
Unfortunately, signatures are large: 20 bytes per chunk. Expected chunk size is a function of how anchors are selected. In our implementation, this size is 16 KB, so signatures are expected to cost about 1.3 MB per GB of stored data. If this cost were paid only once, it might be acceptable. However, a node’s buddy set can change over time as buddies are found to be unreliable or as degrees of overlap change.

Rather than send a full signature, Pastiche nodes send a small, random subset of their signatures called an *abstract*. This is motivated by the following observation: most data on disk belongs to files that are part of a much larger logical entity. For example, a Linux hacker with the kernel source tree has largely the same source tree as others working on the same version. Any machine holding even a small number of random chunks in common with this source tree is likely to hold most of them. Preliminary experiments show that tens of chunkIds—a few hundred bytes—are enough to distinguish good matches from bad ones. This size is similar to that reported by Border for individual web objects [8].

### 3.2 Overlays: Finding a Set of Buddies

All of a node’s buddies should have substantial overlap with it to reduce storage overhead. In addition, most buddies should be nearby to reduce global network load and improve restore performance. However, at least one buddy must be located elsewhere to provide geographic diversity. As a rule of thumb, each Pastiche node maintains five buddies.

Pastiche uses two Pastry overlays to facilitate buddy discovery. One is a standard Pastry overlay organized by network proximity. The other is organized by file system overlap. Every Pastiche node joins a Pastry overlay organized by network distance. Its nodeId is a hash of the machine’s fully-qualified domain name. Once it has joined, the new node picks a random nodeId and routes



This figure depicts how a meta-data chunk is stored on disk. The chunk is stored as a log of file states, where each entry in the log represents the state of the file after the update. Entries are comprised of an attribute block, a time stamp, and a list of constituent chunk handles.

Figure 2: Meta-data Chunk Layout

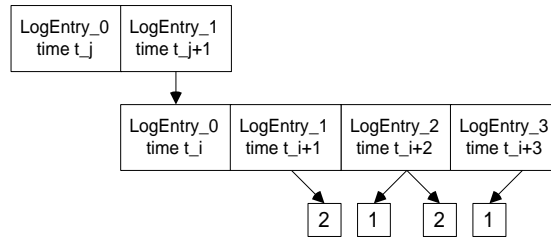
a discovery request to it. The discovery request contains the new node’s abstract. Each node encountered on the route computes its *coverage*—the fraction of chunks in the abstract stored locally—and returns it.

If the initial probe does not generate a sufficient candidate set, the probe process is repeated. Subsequent probes are generated by varying the first digit of the original nodeId. Since Pastry uses prefix routing, each probe will generate sets of candidates disjoint from those already examined. We call this rotating probe a *lighthouse sweep*.

Nodes with common installations should find a sufficient candidate set easily. However, nodes with rare installations will have more difficulty. Nodes that do not find an adequate set during a lighthouse sweep join a second overlay, called the *coverage-rate overlay*. This overlay uses file system overlap rather than network hops as the distance metric. The new node chooses backup buddies from its Pastry neighbor set—the set of nodes encountered during join with the best coverage available.

The use of coverage rate as a distance metric has interesting implications for Pastry. Like network distance, coverage rate does not obey the triangle inequality. Unlike network distance, coverage rate is not symmetric; if A holds all of B’s files, the converse is probably not true. This means that an individual node must build its routing state based on the correct perspective. Likewise, the seeding algorithm must be supplied with the new node’s abstract, so that it can compute coverage from the correct point of view.

It is possible for a malicious node to habitually under- or over-report its coverage. If it under-reports, it can avoid being selected as a buddy. If it over-reports, it can attract unsuspecting clients only to discard their backup



This figure depicts a small skeleton. Each chunk is stored as a log, and each entry in the log has references to other chunks. The top chunk begins empty, and then adds another. The bottom chunk adds a data chunk, appends another to the end, and then removes the first chunk.

Figure 3: Chunk Skeleton

state. Unfortunately, this is possible no matter who computes coverage rates. An honest node can be given a random list of chunkIds as an abstract; such an abstract is unlikely to match anything. Likewise, a malicious node can cache and report abstracts sent by others with commonly-appearing chunkIds, hoping for a false match.

### 3.3 Backup Protocol

A Pastiche node has full control over what, when, and how often to back up. Each discrete backup event is viewed as a single snapshot. Nodes can subscribe to a calendar-based cycle, a landmark-based scheme [43], or any other schedule. Because a machine is responsible for its own archival plan, it keeps a meta-data *skeleton* for each retained snapshot. A file that is not part of the local machine’s current file system, but is part of one or more archived snapshots, has a corresponding meta-data entry stored on the local machine.

The skeleton for all retained snapshots is stored as a collection of persistent, per-file logs, as shown in Figures 2 and 3. The skeleton representing a machine’s current file system state plus all retained snapshots is stored both on the machine and all of its backup buddies.

The state necessary to establish a new snapshot consists of three things: the chunks to be added to the backup store, the list of chunks to be removed, and the meta-data objects in the skeleton that change as a result. We call these the add set, delete set, and meta-data list.

The snapshot process begins by shipping the host’s public key. This key will be associated with any new chunks to validate later requests to delete or replace them. The snapshot node then forwards the chunkIds for elements of the add set. If any of those chunks are not already stored on the buddy, the buddy fetches the chunks from the node.

Next, the node sends the delete list. The snapshot host adds a chunkId to the delete list only if it does not belong to any snapshot the client wishes to retain. The delete list must be signed, and this signature is checked to ensure it matches the public key associated with any chunks scheduled for deletion. Note that deletion is not effected immediately. It is deferred to the end of the snapshot process.

Finally, the snapshot node sends any updated meta-data chunks. Since they may overwrite old meta-data chunks, their chunkIds must also be signed. When all state has been transferred, the host requests a *commit* of the checkpoint. Before responding, the buddy must ensure that all new chunks, changed meta-data objects, and deleted chunkIds are stored persistently. Once that is complete, the buddy can respond, and later apply the new snapshot by performing the appropriate deletions.

The performance of snapshots is not crucial, since they are asynchronous. The only exception is marking chunkstore copy-on-write, which must be done synchronously. However, as with AFS's volume clone operation [23], this is inexpensive. The load induced on a buddy by the backup protocol can be regulated with resource containers [2] or progress-based mechanisms [20]. This load is quantified in Section 5.2.

The snapshot process is restartable. The most expensive phase—shipping new data chunks—makes progress even in the presence of failures, since new chunks are stored as they arrive. After the new snapshot is applied, a faithful buddy will have a complete copy of the meta-data skeleton, plus all data chunks the skeleton names.

### 3.4 Restoration

A Pastiche node retains its archive skeleton, so performing partial restores is straightforward. The node identifies which chunks correspond to the restore request, and obtains them from the nearest buddy.

Recovering the entire machine requires a way to bootstrap the skeleton. To do so, a Pastiche node keeps a copy of its root meta-data object on each member of its network-distance leaf set. When a machine must recover from disaster, it rejoins the distance-based overlay with the same nodeId, which is computed from its host name. It then obtains its root node from one of its leaves, and decrypts it with the key generated from the host's passphrase. Since the root block contains the set of buddies in effect when it was replicated, the node can recover all other state.

### 3.5 Detecting Failure and Malice

A buddy is expected to retain backup snapshots, but is not required to do so. When faced with a sudden disk space crisis, a buddy is free to reclaim space. A buddy may also fail or be connected intermittently, leaving its ability to serve future restore requests in doubt. Finally, a malicious buddy may claim to store chunks without actually doing so.

Pastiche employs a probabilistic mechanism to detect all of these situations. Before taking a new snapshot, each Pastiche node asks buddies for a random subset of chunks from the node's archive. By requesting a modest number of chunks, clients can bound the probability that compromised backup state goes undetected. Savvy users of traditional backup schemes already employ this technique as an end-to-end confirmation of correctness. If a buddy cannot produce data it claims to hold, the client removes it from its buddy list and initiates a search for a replacement. If a buddy has not responded for a significant period of time, the client likewise removes it.

Unfortunately, this request provides only instantaneous assurance; a malicious node can drop chunks after they are requested. Thus, increasing the frequency of requests does not provide increased assurance, while increasing the size of a single request does.

This technique assumes that a malicious party cannot occupy a substantial fraction of the nodeId space, and hence cannot produce collusion between a single host's backup buddies. Defending against such *Sybil attacks* [18] in practice requires some centralized agency to certify identities [9]. Such certification is essential in Pastiche, because nodes can increase their chances of being selected as buddies by falsely over-reporting their coverage rates.

Pastiche leverages the spot-check mechanism to detect snapshots belonging to decommissioned machines. Each buddy knows that its corresponding host does not fully trust it. So, the buddy expects to be probed periodically. If a buddy does not hear from its corresponding host for an exceptionally long period, it can assume that the host has either been decommissioned or re-installed. This decision must be made very conservatively, lest a long-lived failure be mistaken for a voluntary removal.

This lack of trust places some limits on Pastiche's applicability; it is intended primarily for end-user systems for which backup is currently difficult. Back-end repositories are already centrally managed, and adding backup services to them is comparatively simple. However, Pastiche can be extended to such services by choosing one or more backup buddies to be an administered, trusted machine, provided the service's expected workload shows temporal locality. Without such locality, the performance of chunkstore is likely to be poor.

### 3.6 Preventing Greed

A greedy host can aggressively consume space by using storage on many hosts and never retiring any of them. This is the most challenging problem faced by Pastiche. Pastiche needs a distributed quota enforcement mechanism; a node should occupy only as much space as it contributes. We have considered three solutions to this problem, but none are completely satisfactory.

The first solution places nodes into equivalence classes based on the resources they consume. Each node monitors the overall storage costs imposed by its backup clients, and compares these costs to its own usage. Those that are much more space-intensive are ejected, and must search for a more suitable partner. Unfortunately, this mechanism is circumvented by the Sybil attack.

The second approach is to force each node to solve cryptographic puzzles [24] in proportion to the amount of storage it occupies. Forging identities is no defense against this, nor is spreading snapshots across more than the usual number of buddies. However, we dislike this solution for several reasons. First, it adds needless expense to backup, which is antithetical to Pastiche's goals. Second, it trades something other than storage for storage space. Third, not all nodes will have equivalent processing power, so it is difficult to provision the solution properly.

The third approach is to account for space with some form of electronic currency [12]. It is sufficient to use an offline protocol [4]; some amount of double-spending is tolerable as long as abusers can be detected eventually. However, currency accounting requires that backup be *goods atomic* [49]; the exchange of currency and backup state must be an atomic transaction. Adding this complicates Pastiche substantially.

### 3.7 An Alternative Design

Before settling on Pastiche's current approach, we considered an alternative that appears to be a more natural fit to a peer-to-peer substrate. Instead of having a small list of backup buddies, each holding a complete backup, this alternative stores each chunk on the  $K$  Pastry nodes with nodeIds numerically closest to the chunk's identifier. We call this alternative the *fine-grained approach*.

The fine-grained approach has two advantages over Pastiche. First, it ensures that only  $K$  backup copies of a chunk exist anywhere in the network. Second, Pastry takes care of detecting failed or unresponsive hosts, and individual nodes need not keep track of them.

However, the fine-grained approach also has two disadvantages. The first is the loss of network proximity for

most replicas, increasing network load during backup and latency during restoration. Restoration costs can be avoided by caching along the Pastry route taken by backup chunks, but this increases global disk overhead.

The second disadvantage is the difficulty in dealing with malicious nodes. It is much harder for a client to probe for malicious nodes, since the set of nodes containing client state is on the order of the number of chunks. Pastiche trades disk space to reduce network costs and give clients the tools to ensure that their backups are safely stored.

## 4 Implementation

The Pastiche prototype consists of two main components: the chunkstore file system and a backup daemon. Chunkstore is written in C and is implemented primarily in user space for simplicity. The user-space component is called `pclientd`. A small, in-kernel portion implements the `vnode` interface [26], integrating chunkstore with Linux 2.4.18. Pastiche uses the XFS device from Arla [51], an open source AFS implementation, for this in-kernel portion.

Data is stored as individual chunks in an underlying file system. For performance reasons, Pastiche also maintains a cache of contiguous, decrypted copies of recently used files, called *container files*. Our prototype does not yet support whole-machine backup, because we have not implemented booting a kernel from chunkstore.

The XFS device sees only container files, and `pclientd` acts as mediator between the device, the container files, and chunkstore. When an application requests a file that is not in a container, `pclientd` retrieves the meta-data chunk for that file from chunkstore and uses it to form a contiguous container file. `pclientd` then returns the inode of the container file to the device, and subsequent operations are applied to the container. The container file cache is managed with LRU replacement, given a maximum size.

`pclientd` is notified of each `close`. If the corresponding file is dirty, it is scheduled for chunking. Chunking is deferred for 30 seconds, to avoid needless overhead for short-lived files [50]. We implemented convergent encryption using the `openssl-0.9.7-beta3` cryptographic library. Each chunk was encrypted using a 128-bit key and the AES stream cipher [17].

Container files restore the parity between logical and on-disk proximity that storing chunks individually eliminates. However, storing chunks individually still induces some storage overhead. By storing each chunk separately, Pastiche files will yield more internal fragmentation than if they had been stored contiguously. Recall that content-based indexing generates chunks by exam-

ining the lower  $k$  bits in a Rabin fingerprint; if these bits match some target value, that offset is marked as a chunk boundary. On average, one would expect to lose half of a disk block per chunk. So, we set  $k$  to 14, giving an expected chunk size of 16KB and expected fragmentation overhead to 3.1%.

Meta-data chunks are stored as a log of updates to the file. Each time a file is re-chunked, the list of its constituent chunks is appended to the log. Deletion is represented with a terminal log entry. `pclientd` only appends to these logs, and thus never removes a chunk from chunkstore.

The backup daemon, called `backupd`, is written in C and uses the `rpc2` remote procedure call package for communication [44]. It acts as both the backup server and client. The server manages remote requests for storage and restoration, while the client supervises selection of buddies and snapshots. Additionally, `backupd` cleans meta-data logs and reaps deleted chunks.

`backupd` communicates with `pclientd` through file locking of on-disk chunks. This is simple and can be efficient, since `backupd` need not hold all locks to guarantee a consistent snapshot. Once the root meta-data chunk is read, all reachable chunks are guaranteed to remain reachable, since none of them will be deleted by `pclientd`. Some meta-data chunks may still become tainted [25] with additional log entries. However, these entries can be detected via timestamps during backup and restore, rendering their inclusion in a snapshot harmless.

We also provide several utilities to allow users to manage the file system: `forcesnap` forces `backupd` to take a system snapshot immediately, `forcechunk` forces `pclientd` to chunk all files in its queue immediately, and `rfile` restores a file or subtree to a previous state. Each utility communicates with `pclientd` and `backupd` through Unix domain sockets.

## 5 Evaluation

In evaluating our prototype, we set out to answer the following questions:

- What is the performance of the file system? Which operations perform well and which perform badly?
- How long do backups and restores take?
- How large must fingerprints be? Is the lighthouse sweep able to find buddies?
- Does the coverage-rate overlay yield suitable backup buddies?
- Are the costs to detect malicious nodes reasonable?

All experiments were run on machines with a 550 MHz Pentium III Xeon processor, 256MB of memory, and a 10k RPM SCSI Ultra wide disk, with 4.7 ms seek time, 3.0 ms rotational latency, and 41 MB/s peak throughput.

AB phase	ext2fs	chunkstore
<code>mkdir</code>	1.23 (0.04)	1.03 (0.05)
<code>cp</code>	3.47 (0.28)	6.26 (0.16)
<code>scandir</code>	0.0 (0)	0.03 (0)
<code>cat</code>	1.75 (0.02)	2.23 (0.02)
<code>make</code>	38.62 (0.45)	38.88 (0.5)
total	45.08 (0.39)	48.43 (0.58)

This figure presents the results of a modified Andrew Benchmark. Times are reported in seconds, and standard deviations are given in parentheses.

Figure 4: Andrew Benchmark

Task	ext2fs	chunkstore
<code>wide create</code>	2.44 (0.06)	6.99 (0.02)
<code>wide mkdir</code>	2.30 (0.02)	6.31 (0.03)
<code>deep mkdir</code>	4.07 (0.02)	5.64 (0.01)
<code>bulk xfer</code>	12.79 (0.01)	12.75 (0.02)

This figure presents the results of file creation and I/O throughput benchmarks. Times are reported in seconds, and standard deviations are given in parentheses.

Figure 5: Micro-benchmarks

### 5.1 Performance

What is the overhead induced by chunkstore? To answer this, we compare the performance of chunkstore to the underlying, native file system, `ext2fs`. We measure this overhead with a modified Andrew Benchmark [23]. Our benchmark is identical to the original in form, but uses the `apache 1.3.26` source tree. This source tree is 9.6MB in size; when compiled, the tree occupies 12MB.

We ran five trials; the results are shown in Figure 4. While the `make` step is not I/O bound, it does experience slight overhead. This is due in part to the cost of computing the Rabin fingerprints of the copied tree and the extra cost of creating and deleting files. The copied data is scheduled to be chunked when written, and 30 seconds later—during the `make` step—chunking begins.

The total overhead of 7.4% is reasonable, though the `copy` phase is expensive; it takes 80% longer in chunkstore. We believe that this overhead is due to excess meta-data management in chunkstore, rather than limits on peak I/O throughput. To confirm our hypothesis, we performed several micro-benchmarks to isolate the operations involved in copying a source tree - writing data and creating files.

To examine chunkstore’s performance when creating files and directories, we ran three different experiments—`wide create`, `wide mkdir`, and `deep mkdir`. In `wide create`, 1000 new files were created in the same directory. In `wide mkdir` 1000 new directories were created in the same directory,

Task	time
cp	6.26 (0.07)
backup	6.55 (0.01)
rm	1.24 (0.01)
restore	5.54 (0.07)
nfs cp	3.76 (0.16)

This figure presents the results of the backup and restore experiment. Times are reported in seconds, and standard deviations are given in parentheses.

Figure 6: Backup and Restore

and in `deep mkdir` 1000 new directories were made recursively inside of one another. We again ran five trials of each; the results are in Figure 5.

`wide create` and `wide mkdir` each ran about 186% and 174% slower than `ext2fs`, respectively, while `deep mkdir` ran about 38% slower. Chunkstore’s poor performance is due to meta-data chunk and container file maintenance. When chunkstore creates a file, it must update or create three files: a new meta-data chunk, a new container file, and the parent meta-data chunk.

The `deep mkdir` experiment shows that the number of entries in the parent directory is also significant. This is because of the way directory entries are laid out in the meta-data chunks and the container files. In both cases, directory entries are stored in a linear array. Our current implementation rewrites the entire list to the container file and chunk whenever a new entry is added. During `deep mkdir`, there is only ever one entry in the list, which makes creating a file faster.

It is also interesting to note that `wide mkdir` is somewhat faster than `wide create`. The reason for this is related to how the in-kernel XFS device handles file and directory creation. When a regular file is created, the XFS device makes an extra upcall to `pclientd` to close the newly created file, and does not make this call when a directory is created.

To further verify that I/O throughput was not responsible for chunkstore’s slow `copy` phase in the modified Andrew Benchmark, we also ran a `bulk xfer` experiment. In this experiment, a new file was created, 256MB of data were written to it, and then the file was closed. As before, we ran five trials; the results are in Figure 5. Chunkstore and `ext2fs` performed within 1% of each other, meaning that their I/O throughput are statistically identical.

## 5.2 Backing Up and Recovering a File System

To determine the performance of our backup and restore utilities, we applied them to a file system consisting of the `openssl-0.9.7-beta3` source tree. This 13.4MB tree

of 1641 files and 109 directories is stored in Pastiche as 4004 chunks.

Each of five trials consisted of four phases - copying the source tree into the file system, sending it to a backup buddy, removing the local source tree, and restoring the source tree from the backup buddy. Pastiche’s backup and restore performs comparably to the time to copy the source tree over NFS. The results are in Figure 6.

It should also be noted that the demand on resources the buddy experiences while carrying out backup and restore is very bursty. During the five trials, `backupd` used a maximum of 8MB of memory, averaged 12 disk transfers/sec with a maximum of 414 transfers/sec, and averaged a 70% idle CPU with a minimum of 13%.

## 5.3 Finding Buddies

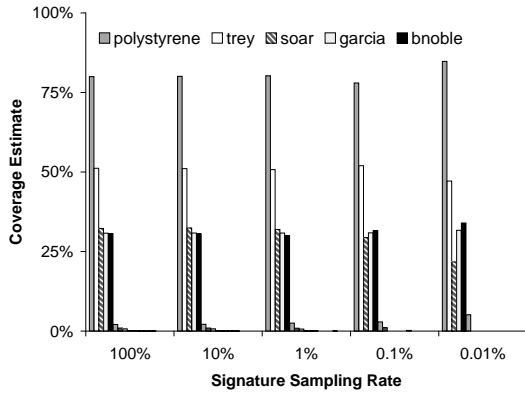
Next we turn our attention to the buddy discovery process. There are three questions to answer. First, how large must an abstract be to discriminate good buddy candidates from bad ones? Second, how effective is the lighthouse sweep in discovering buddies? Third, how effective is the coverage-rate overlay in discovering buddies? To answer the first question, we took the signatures of seventeen machines at Michigan. These machines run Windows, Linux, Solaris, and various flavors of BSD. We also took the signatures of two freshly installed machines.

The first ran Windows 98 with an Office 2000 Professional installation, but without any service packs applied. This machine held roughly 90 thousand chunks<sup>1</sup>. The second was a Linux machine running a Debian `unstable` release, configured as a conventional workstation with development and document processing tools. This machine held approximately 270 thousand chunks. We chose this machine as a worst case. Some of our comparison machines are Debian, but only one runs the `unstable` distribution. This distribution changes quickly, and this machine is updated infrequently.

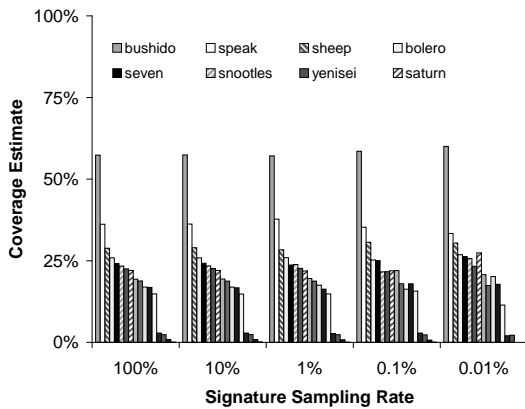
We computed the actual coverage for each of these machines given full signatures. To estimate the impact of smaller abstracts on coverage estimates, we took uniform random samples of the signature at rates of 10%, 1%, 0.1%, and 0.01%; there are six samples at each rate.

The results for the Windows 98 machine are plotted in Figure 7(a). The  $x$  axis gives sampling rate, and the  $y$  axis shows coverage rate. The 100% “sample” shows exact coverage; each of the others is an estimate given a smaller sample. Each group of bars represents the coverage estimate for each of the seventeen hosts. Within each group, the hosts are sorted by actual coverage rate,

<sup>1</sup>For this experiment, we used a smaller expected chunk size of 4KB; Pastiche’s larger chunks may require slightly larger sampling rates.



(a) Windows 98/Office 2000



(b) Debian Developer

Figure 7: Varying Abstract Size

from highest to lowest. The top five matches are identified in the legend. `polystyrene` is a Win98 machine running Office 2000, with all relevant service packs and security updates applied.

The estimates are surprisingly independent of sample size; the lowest rate produces abstracts of around 10 chunkIds. Only `soar`'s estimate changes appreciably. However, its coverage rate is comparable to `garcia`'s and `bnoble`'s; choosing either of the latter in favor of `soar` is of no consequence.

Figure 7(b) shows the results for our Linux machine, the top eight matches are identified in the legend. The overall match rates are lower, but machines with other distributions still have substantial matches. As with the Windows 98 host, coverage estimates do not change materially as abstract sizes go down. Interestingly, `bnoble`, a Windows 2000 machine, has a coverage rate for this Debian machine of almost 15%. This is because `bnoble` also has a `stable` release of Debian, installed in a VMware virtual machine; the VMware disk image is stored as a regular file in Windows. Ordinarily, files

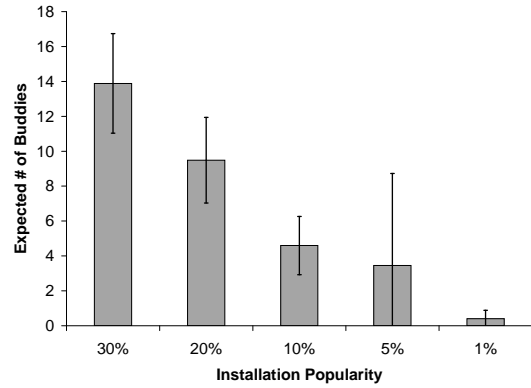


Figure 8: Expected Number of Buddies

form implicit chunk boundaries in content-based indexing. When viewed from the windows host, all of these file boundaries disappear. Despite this, content-based indexing is still able to find substantial overlap.

Small abstracts are effective only if they are delivered to a host that can provide good coverage. We conducted a simulation to determine how effectively lighthouse sweeps find useful buddies. This simulation uses SimPastry [29], a Pastry simulation/visualization tool.

For the simulation, we populated a graph with 50 thousand Pastiche nodes drawn from a distribution of 11 types. 30% of all nodes are the first type, types two and three each comprise 20% of all nodes, types four and five each comprise 10%, type six comprises 5%, and types seven through eleven each represent 1% of the population.

We simulated 25 different Pastry networks under these conditions. For each network, we randomly selected 1000 nodes of each type, and performed a lighthouse sweep from that node, counting the number of hosts of identical type found during the sweep. The results are shown in Figure 8. Each bar gives the average number of matches found per sweep for each category of popularity; the error bars show one standard deviation.

As expected, common nodes with representation of 10% or higher should find an adequate number of buddies on the distance overlay. Those with lower popularity will need to join the coverage-rate overlay as well. We built a Pastry simulator to determine the effectiveness of this network. Our experiments involved 10,000 nodes. Each node was assigned to one of a thousand species, one of a hundred genera, and one of ten orders. Nodes of the same order share 20% of their content; nodes of the same genus, 30%; and nodes of the same species, 70%. Only nodes of the same species can serve as backup buddies for one another.

The results of our simulations are in Figure 9. The  $x$  axis give the size of the neighborhood set, and the  $y$

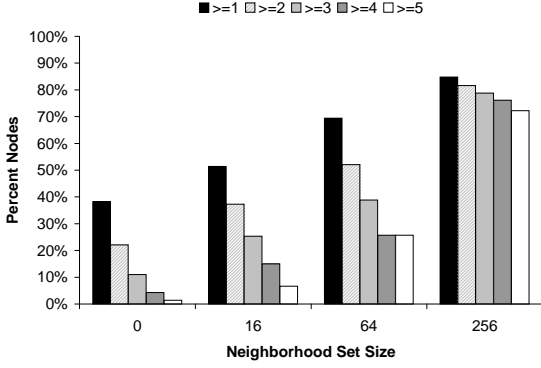


Figure 9: Coverage-rate Simulation Results

axis shows the percent of all nodes who found a given number of buddies. We ran four series of trials, varying the size of the neighborhood set. We found that for a neighborhood set size of 256, 85% were able to find at least one buddy in its routing table, and 72% were able to find at least 5.

The results show that most nodes should be able to find buddies in the coverage-rate overlay. It also shows how important a role the neighborhood set plays in locating buddies. Increasing the neighborhood set from 0 to 256 increases the percent of nodes who can find at least one buddy from 38% to 85%; the percent of nodes who can find at least 5 increases from 1% to 72%.

#### 5.4 Determining Query Size

Backup buddies can drop chunks, either in error or maliciously. If the same chunk is dropped at all replicas, the backup state is said to be *corrupted*. A node can be certain that its state is not corrupted by requesting all  $c$  chunks, but this is clearly too expensive. Instead, Pastiche nodes query just enough chunks,  $q$ , to be assured with some probability  $p$  that corrupted state will be detected if it exists.

We assume that replicas cannot collude to agree on a specific chunk to drop. Instead, each of  $n$  replicas drops chunks at some rate  $r$ . A Pastiche node must set  $q$  so that the probability of drops causing corruption and going undetected is less than or equal to  $p$ .

If  $r$  is zero, then the chance of corruption is also zero, and the problem is solved trivially. On the other hand, if  $r$  is one, then a query of one chunk is guaranteed to detect corruption. However, some intermediate values of  $r$  require queries of more chunks.

The analysis proceeds in two parts. First, we compute  $p_c$ , the probability of corruption. Second, we compute  $p_u$ , the probability that dropped chunks go undetected. These are conditionally independent for a given  $r$ , so

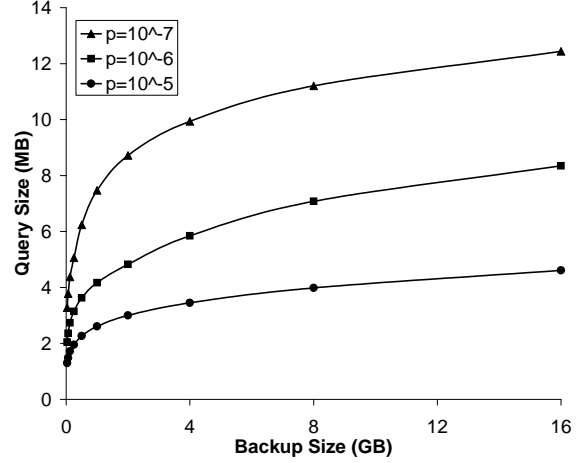


Figure 10: Growth of Query Size

their product expresses the event whose probability we wish to bound by  $p$ .

If each replica drops each chunk with probability  $r$ , then the chance that a chunk is dropped at all replicas is  $r^n$ , and the chance that it exists on at least one is  $1 - r^n$ . There are  $c$  chunks, so the chance that all of them exist on at least one replica is  $(1 - r^n)^c$ . Therefore, the chance that at least one of them does *not* exist on *all* replicas—the chance of corruption—is:

$$p_c = 1 - (1 - r^n)^c. \quad (1)$$

Suppose a node asks a buddy for a single chunk. The chance that the buddy can supply it is  $1 - r$ . If the node asks for  $q$  chunks simultaneously, then the chance that the buddy can respond successfully—the chance that dropped chunks go undetected—is

$$p_u = (1 - r)^q. \quad (2)$$

We want to bound the product of these to minimize the probability of undiscovered corruption:

$$[1 - (1 - r^n)^c](1 - r)^q \leq p. \quad (3)$$

Solving for  $q$ , we get

$$q \geq \log_{1-r} \left( \frac{p}{1 - (1 - r^n)^c} \right). \quad (4)$$

Since  $r$  must take one of  $c + 1$  discrete values, it is feasible to compute the maximum  $q$  over all possible values of  $r$ , given  $n$ ,  $c$ , and  $p$ .

The resulting queries grow very slowly with respect to backup size, as shown in Figure 10. This figure shows total backup size on the  $x$  axis and computed query size on the  $y$  axis, assuming an average chunk size of 16KB

and five replicas. The query sizes are computed for five buddies, with three values of  $p$ :  $10^{-5}$ ,  $10^{-6}$ , and  $10^{-7}$ . Even for high degrees of assurance, the query costs are modest.

## 6 Related Work

Backup is critical, yet there is a surprisingly small amount of literature on the topic. Most work focuses on centralized backup of large installations [28, 36]; Chervenak provides a survey of a number of different backup systems [13]. Current commercial systems, such as Veritas' NetBackup, IBM's Tivoli, and Connected's remote backup service, also focus on large, centrally-managed repositories.

Several projects have suggested the use of peer-to-peer routing and object storage systems as a substrate for backup, including Chord [46], Freenet [14], and Pastry [41]. File systems built on them, such as PAST [42] and CFS [16], provide protection against machine failure. However, they do not protect against human error, nor do they provide the ability to retrieve prior versions of files once replaced. OceanStore [40], does provide these benefits, but the decision of which versions to retire rests with the utility, not its clients.

The pStore cooperative backup system [3], built on top of Chord, stores individual objects on a number of nodes, rather than storing the entire set of objects on a number of nodes. However, it does not exploit inter-host sharing, nor does it address the problem of hosts falsely claiming to store data. Elnikety presents a cooperative backup scheme [21] that requests random blocks from partners, but assumes that partners either drop all or none of the archived state.

A number of systems exploit duplicate data across files and machines to minimize the costs of archival storage. The Single Instance Store [5] detects and coalesces duplicate files, while Venti [38] divides files into fixed-size blocks and hashes those to find duplicate data. Neither of these approaches can take advantage of small edits that move data within a file, as content-based indexing does [27, 30]. Other sophisticated techniques for detecting such changes exist [1, 48], but must be run on pairs of files that are assumed to have overlap.

Broder provides a mathematical foundation for detecting similarity and inclusion based on sketches [8], similar to Pastiche's abstracts. Sketches of a few hundred bytes are able to find similarities among single documents on the web [7]. Pastiche extends this result, to find similarities between entire disks.

Rather than exploit redundancy, one can instead turn to the use of erasure codes [35] to stripe data across several replicas. Such codes allow for low-overhead replication, and are tolerant of the failure of one or

more replicas; they are employed by Myriad [11], OceanStore [40], and Elnikety [21]. Their main shortcoming, compared to our simpler scheme, is that they require the participation of more than one node for restore.

AFS [23], Plan 9 [37], and WAFL [22] expose a snapshot primitive for a variety of purposes, including backup. Typically, snapshots are used to stage data to archival media other than disk. SnapMirror [34] leverages WAFL's snapshot mechanism to provide fine-grained, remote disk mirroring with low overhead.

## 7 Conclusion

Backup is tedious and expensive. Embarrassingly, the authors' own workstations are not backed up. Only their user data, stored on a distributed file system, is backed up regularly.

Pastiche enables automatic backup with no administrative costs, requiring only excess disk capacity among a set of cooperating peers. Since Pastiche matches nodes who have significant data in common, this excess capacity can be modest. Peers are selected through the use of two peer-to-peer overlay networks, one organized by network distance, the other by degree of data held in common. The self-organizing nature of these overlays, combined with mechanisms to detect failed or malicious peers, obviates the need for administrative intervention.

Evaluation of our Pastiche prototype demonstrates that this service does not penalize file system performance unduly. Simulations confirm the effectiveness of node discovery, and analysis shows that detecting malicious hosts requires only modest resources. Pastiche promises to lower the barriers to backup so that all data can be protected, not just that judged worthy of the expense and burden of current schemes.

## Acknowledgements

The authors wish to thank Mark Brehob for his assistance with the analysis in Section 5.4. Jason Flinn, Jim Gray, and the anonymous reviewers provided many helpful comments.

This work is supported in part by the Intel Corporation; Novell, Inc.; the National Science Foundation under grant CCR-0208740; and the Defense Advanced Projects Agency (DARPA) and Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either

expressed or implied, of the Intel Corporation; Novell, Inc.; the National Science Foundation; the Defense Advanced Research Projects Agency (DARPA); the Air Force Research Laboratory; or the U.S. Government.

## References

- [1] M. Ajtai, R. Burns, R Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery*, to appear.
- [2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, February 1999.
- [3] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. Unpublished report, MIT Laboratory for Computer Science, December 2001.
- [4] M. Blaze, J. Ioannidis, and A. Keromytis. Offline micro-payments without trusted hardware. In *Proceedings of the Fifth Annual Conference on Financial Cryptography*, Cayman Islands, BWI, February 2001.
- [5] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24, Seattle, WA, August 2000.
- [6] 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. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 34–43, Santa Clara, CA, June 2000.
- [7] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International World-Wide Web Conference*, pages 391–401, Santa Clara, CA, April 1997.
- [8] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES*, pages 21–29, Salerno, Italy, June 1997. Published in 1998.
- [9] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.
- [10] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Submitted for publication.
- [11] F. Chang, M. Ji, S. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. Myriad: Cost-effective disaster tolerance. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 103–116, Monterey, CA, January 2002.
- [12] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of Crypto '82*, pages 199–203, August 1982.
- [13] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: A survey of backup techniques. In *Proceedings of the Joint NASA and IEEE Mass Storage Conference*, March 1998.
- [14] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting fee expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.
- [15] Connected Corporation. The 60% you're missing: Preventing data loss through PC management. White paper, Farmingham, MA, 2002.
- [16] 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*, pages 202–215, Banff, Canada, October 2001.
- [17] J. Daemen and V. Rijmen. AES proposal: Rijndael. Advanced Encryption Standard Submission, 2nd version, March 1999.
- [18] J. R. Douceur. The Sybil attack. In *1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002.
- [19] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 59–70, Atlanta, GA, May 1999.
- [20] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 247–260, Kiawah Island Resort, SC, December 1999.
- [21] S. Elnikety, M. Lillibridge, M. Burrows, and W. Zwaenepoel. Cooperative backup system. In *The USENIX Conference on File and Storage Technologies*, Monterey, CA, January 2002. Work-in-progress report.
- [22] D. Hitz, J. Lau, and M. A. Malcom. File system design for an NFS file server appliance. In *Proceedings USENIX Winter Technical Conference*, pages 235–246, San Francisco, CA, January 1994.
- [23] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [24] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–165, San Diego, CA, February 1999.
- [25] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance

- and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Saint Malo, France, October 1997.
- [26] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Association Summer Conference Proceedings*, pages 238–247, Atlanta, GA, June 1986.
- [27] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Conference*, pages 1–10, San Francisco, CA, January 1994.
- [28] E. Melski. Burt: the backup and recovery tool. In *Proceedings of LISA'99*, pages 207–217, Seattle, WA, November 1999.
- [29] Microsoft Corporation. SimPastry. <http://www.research.microsoft.com/~antr/Pastry/download.htm>.
- [30] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Banff, Canada, October 2001.
- [31] National Institute of Standards and Technology. Computer data authentication. FIPS Publication #113, May 1985.
- [32] National Institute of Standards and Technology. Secure hash standard. FIPS Publication #180-1, April 1997.
- [33] Network Appliance. NetApp unveils first nearstore release. *Computer Reseller News*, page 33, March 25, 2002.
- [34] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 117–129, Monterey, CA, January 2002.
- [35] W. W. Peterson and E. J. Weldon. *Error-correcting Codes*. The MIT Press, 1972.
- [36] W. C. Preston. Using Gigabit Ethernet to backup six Terabytes. In *Proceedings of LISA'98*, pages 87–95, Boston, MA, December 1998.
- [37] S. Quinlan. A cache WORM file system. *Software—Practice and Experience*, 21(12):1289–1299, December 1991.
- [38] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 89–102, Monterey, CA, January 2002.
- [39] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [40] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, September 2001.
- [41] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, November 2001.
- [42] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, Banff, Canada, October 2001.
- [43] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofler. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, Kiawah Island Resort, SC, December 1999.
- [44] M. Satyanarayanan. *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon University, October 1991.
- [45] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, May 1996.
- [46] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, pages 149–160, San Diego, CA, August 2001.
- [47] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 165–179, San Diego, CA, October 2000.
- [48] A. Tridgell. *Efficient algorithms for sorting and synchronization*. PhD thesis, The Australian National University, 1999.
- [49] J. D. Tygar, A. Gupta, O. Shmueli, and J. Widom. Atomicity versus anonymity: Distributed transactions for electronic commerce. In *Proceedings of the 24th Annual International Conference on Very Large Data Bases*, pages 1–12, New York, NY, August 1998.
- [50] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island Resort, SC, December 1999.
- [51] A. Westerlund and J. Danielsson. Arla—a free afs client. In *Proceedings of 1998 USENIX, Freenix track*, New Orleans, LA, June 1998.